ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šamánek** Jméno: **Filip** Osobní číslo: **393039**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Datové vědy**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Implementace algoritmů FOL pro knihovnu SymPy

Název diplomové práce anglicky:

Implementation of FOL algorithms for the SymPy library

Pokyny pro vypracování:

Vypracujte Python implementaci alespoň těchto algoritmů pro logiku prvního řádu: subsumpce, prenexová normální forma, Skolemizace, least general generalization, rezoluce s rovností, zploštění klauzulí. Seznamte se s problematikou, proveďte analýzu a navrhněte implementaci v Pythonu. Vybrané algoritmy implementujte a otestujte. Vyhodnoťte kvalitu implementace vhodnými kritérii (např. srovnáni rychlosti s existujícími implementacemi FOL rezoluce v Pythonu).
Implementace by měla organicky doplňovat knihovnu SymPy a by měla mít asymptoticky optimální složitost. Ve vhodných případech je možno použít i existující implementace, pokud to licence umožňuje. Implementace heuristik není vyžadována, ale je vhodné provést jejich analýzu. Implementaci připravte na přijetí kódu do oficiálních repozitářů SymPy.

Seznam doporučené literatury:

[1] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 https://doi.org/10.7717/peerj-cs.103
[2] G.Plotkin. A note on inductive generalization. Edinburgh University Press, 1970. ISBN 80-01-02095-9
[3] Robinson, J. Alan (1965). 'A Machine-Oriented Logic Based on the Resolution Principle'. Journal of the ACM. 12 (1): 23–41. doi:10.1145/321250.321253.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Božena Mannová, Ph.D., kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **20.09.2020**

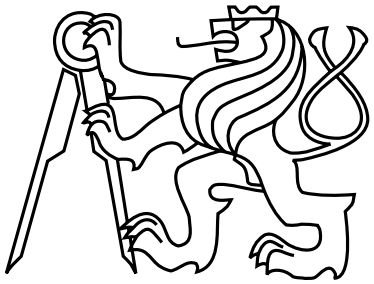
Ing. Božena Mannová, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání | Podpis studenta

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Implementace algoritmů FOL pro knihovnu SymPy

Bc. Filip Šamánek

Vedoucí práce: ING. BOŽENA MANNOVÁ, PH.D.

Studijní program: Otevřená informatika, Magisterský

Obor: Datové vědy

24. května 2019

Poděkování

Rád bych poděkoval své vedoucí, Ing. Božena Mannová, Ph.D., za radu a vedení. Kdykoliv jsem potřeboval poradit, vždy mi pomohla najít odpověď.

Chtěl bych také vyjádřit svou vděčnost mé rodině a přátelům, kteří nemohli v průběhu mého studia více podporovat.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 24. 5. 2019 .....

Abstract

This diploma thesis concerned with the implementation of basic algorithms for first order logic. The algorithms are implemented in Python language, particularly they will be part of the SymPy library.

The majority of the thesis is focused on analysis of individual algorithms and it's design of implementation to SymPy library. The method of implementation to SymPy library, aiming to respect the SymPy library principles, so the solution will become integral part of that library. The integration to library SymPy will allow to access the algorithm for first order logic by wider area of users.

Abstrakt

Tato diplomová práce se zabývá implementací základních algoritmů pro logiku prvního řádu. Algoritmy jsou implementovány v jazyce Python, konkretně budou součásti knihovny SymPy.

Velká část práce se zabývá analýzou jednotlivých algoritmů a návrhem jejich implementace do knihovny SymPy. Způsob implementace do knihovny se snaží o respektovaní jejích principu tak, aby se řešení stalo integrální součásti této knihovny. Integrace do knihovny SymPy umožní zpřístupněni algoritmů pro logiku prvního řádu širšímu okruhu uživatelů.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Motivace

Logika prvního řádu se využívá v matematice, filozofii, lingvistice a informatice. Jedná se o formální systém vhodný k reprezentaci znalostí a teorémů. Proto má velké využití při dokazování platnosti výroků a v umělé inteligenci. Příkladem použití může být převod přirozeného textu do logiky prvního řádu a následné strojové učení z takto převedených znalostí.

Při řešení některých domácích úkolů v průběhu studia mi chyběla knihovna pro Python, která by umožňovala práci s logikou prvního řádu. Požadavek na to, aby byla knihovna v Pythonu je z důvodu, že je mi tento jazyk blízký a zároveň to je jeden z jazyků ve kterém můžeme vypracovat většinu domácích úkolů ve škole. Další z požadavků byl, aby se jednalo o nějakou větší, známou a rozšířenou knihovnu. Tento požadavek se velmi špatně specifikuje, je to spíše o pocitu každého programátora. Na druhou stranu to pro mě byl důležitý požadavek, protože zpravidla platí, že čím je knihovna větší, tím má lepší dokumentaci, jsou v ní opravovány chyby, je udržována aktuální a hlavně k ní existuje dostatečně velká komunita, která může poradit s řešením problémů.

Bohužel takovou knihovnu jsem pro Python nenašel a proto jsem si zvolil jako téma mé diplomové práce implementaci algoritmů pro logiku prvního řádu do knihovny SymPy. Knihovna SymPy splňuje moje požadavky a zároveň jsem se setkal s touto knihovnou při řešení jiných úkolů, takže není pro mě neznámá a algoritmy pro logiku prvního řádu do ní zapadají.

1.2 Cíle

Cílem této práce je implementovat základní algoritmy pro logiku prvního řádu. Konkrétně se jedná o algoritmy: subsumpce, prenexová normální forma, skolemizace, least general generalization, rezoluce s rovností, zploštění klauzulí. Tyto algoritmy budou implementovány v jazyce Python jako integralní součást knihovny SymPy. Další cíl je přijetí mého kódu do oficiálního repozitáře knihovny SymPy.

Kapitola 2

Knihovna SymPy

Popis knihovny SymPy vychází z článku *"SymPy: symbolic computing in Python"*[11] z dokumentace knihovny SymPy [13] Knihovna SymPy je napsána čistě v jazyce Python a na rozdíl od spousty ostatních počítačových algebraických systémů nemá vlastní jazyk, ale využívá přetěžování operátorů Pythonu. SymPy využívá embedded domain specific language paradigma navržené Hudak (1998)[7].
Licence: Tříbodová BSD licence (BSD = Berkeley Software Distribution)
Podporované verze Pythonu: 2.7, 3.4, 3.5, 3.6 a PyPy

2.1 Příklady použití

Výpočet limity $\lim_{n\to\infty}(x * \sin(\frac{1}{x}))$

```
>>> from sympy import *
>>> x = Symbol('x')
>>> limit(x*sin(1/x), x, oo)
1
```

Derivace funkce $\sin(x) * e^x$

```
>>> diff(sin(x)*exp(x), x)
exp(x)*sin(x) + exp(x)*cos(x)
```

2.2 Architektura knihovny

2.2.1 Datová struktura

Matematické výrazy jsou ukládány do stromové datové struktury s uspořádanými potomky. Například x+y je uloženo jako tři uzly "+", "x", "y" kde uzly "x" a "y" jsou uspořádaní potomci uzlu "+". Jednotlivé funkce pro manipulaci s matematickými výrazy mají na vstupu

tuto datovou strukturu. S touto datovou strukturou pracují a vrací výsledek zase v této datové struktuře.

Všechny výrazy v SymPy jsou immutable, což zjednodušuje práci s výrazy a umožňuje jejich hašování (hashing). Tím pádem lze používat výraz jako klíč v Python datové struktuře slovník (dictionaries). Zároveň to umožňuje kešování (caching) jednotlivých výrazů v SymPy.

2.2.2 Objektová struktura

V knihovně se nepoužívají standardní datové typy Pythonu, jako je například boolean. Místo toho knihovna obsahuje vlastní třídy, které používa pro práci s boolean, a na který je boolean převeden. Stejně tak obsahuje vlastní konstanty se kterými pracuje (například pro True a False má vlastní konstanty). To umožňuje přetěžovat metody pro matematické a další operace.

V SymPy je každý symbolický výraz instancí třídy *Basic*. Třída *Basic* je superclass všech typů v knihovně SymPy. Jednotliví potomci uzlu jsou uloženy v atributu *args*. Listy stromu mají atribut *args* prázdný.

Většina tříd v SymPy jsou zároveň potomci třídy *Expr*, například *Add*, Mul, *Symbol*, ... Jsou tu ale i výjimky, například třída *And* je potomek třídy *Basic*, ale není potomek třídy *Expr*.

Třída *Expr* je základní třída pro jakékoliv algebraické výrazy. Všechny třídy, které vyžadují implementaci aritmetických operátorů, by měli být potomkem této třídy.

Další základní třída v SymPy je třída *Function*, která je potomek třídy *Expr* a třídy *Application*. Slouží především k definování funkcí jako je $sin(x), log(x), ...$ Tato třída slouží jako kontruktor pro nedefinované funkce.

Třída *Application* slouží jako základní třída pro všechny aplikované funkce.

2.2.3 Matematické operace s výrazy

Vzhledem k tomu, že Python umožňuje přetěžování operátorů, tak můžeme matematické výrazy zapisovat například:

```
>>> x*y + 2
```

Tento výraz je v Pythonu pomocí knihovny SymPy přeložen jako

```
>>> x.__mul__(y).__add__(2)
```

, kde proměnné x a y obsahují instanci třídy *Symbol* a 2 je typu *int*. Typ *int* je převeden na SymPy typ *Integer*.

Standardní chování této knihovny při porovnávání dvou výrazů pomocí == je porovnávání stromu výrazů a ne matematická rovnost. Například:

```
(x+1)**2 == x**2 + 2*x + 1
```

vrátí false, protože výrazy mají různou stromovou reprezentaci.

2.3 Výkonost

Výkonost (performance) knihovny SymPy není optimální a to hlavně z důvodu, že je psaná čistě v jazyce Python. Z toho důvodu vznikl projekt SymEngine [12], což je knihovna napsaná čistě v $C++$. Pro SymEngine existují wrappery do různých jazyků jako jsou Python, Ruby, Julia .... Knihovna Sympy ve většině případů v porovnání s komerčními, konkurenčními produkty vychází hůře. Její výkon je ale přesto dostačující pro řešení většiny problémů. Její limity záleží na konkretním hardwaru počítače. Na běžném novém počítači by její limity měly být kolem 10^4 - 10^6 symbolů. [1]

[1] Tyto údaje jsem převzal z článku $SymPy : symboliccomputinginPython$[11].

Kapitola 3

Logika prvního řádu

Definice pojmů logiky prvního řádu a jejich vysvětleni je citováno z *Matematická logika, Marie Demlová [10]*, *Matematical logic, Marie Demlová [4]*, *Symbolic Machine Learning, Filip Železný, Jiří Kléma [6]*

3.1 Syntaxe logiky prvního řádu

3.1.1 Jazyk predikátové logiky

Jazyk predikátové logiky se skládá z:

1. Logických symbolů:
   (a) Spočetné množiny individuálních proměnných: **Var** $= \{x, y, ..., x_1, x_2, ..., x_n\}$
   (b) Výrokových logických spojek: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
   (c) Obecného kvantifikátoru $\forall$, a existenčního kvantifikátoru $\exists$
2. Speciálních symbolů:
   (a) Množiny **Pred** predikátových symbolů (nesmí být prázdná)
   (b) Množiny **Kons** konstantních symbolů (může být prázdná)
   (c) Množiny **Func** funkčních symbolů (může být prázdná)
3. Pomocných symbolů, jako jsou závorky „(, [,) ,]“ a čárka „,“

Predikátové symboly se budou dále v textu většinou značit velkými písmeny, tj. např. $P, Q, R, ..., P1, P2, ...$; konstantní symboly malými písmeny ze začátku abecedy, tj. $a, b, c, ..., a1, ...,$; proměnné malými písmeny z konce abecedy tj. $x, y, z, ...x_1, x_2, ..., z_n$ a funkční symboly většinou $f, g, h, ..., f1, f2, ...$ Formule predikátové logiky budu označovány malými řeckými písmeny. Kdykoli bude se od těchto konvencí odchýlí, tak v textu na to bude upozorněno.

3.1.2 Arita

Pro každý predikátový i funkční symbol máme dáno přirozené číslo n větší nebo rovno 1, které nám udává, kolika objektů se daný predikát týká, nebo kolika proměnných je daný funkční symbol. Tomuto číslu říkáme arita predikátového symbolu nebo funkčního symbolu. Někteří autoři umožňují definovat i funkční symboly arity 0, což jsou konstanty.

3.1.3 Termy

1. Každá proměnná a každý konstantní symbol je term.
2. Jestliže f je funkční symbol arity n a $t_1, t_2, ..., t_n$ jsou termy, pak $f(t_1, t_2, ..., t_n)$ je také term.
3. Nic, co nevzniklo konečným použitím pravidel 1 a 2, není term.

3.1.4 Atomické formule

Atomická formule je predikátový symbol P aplikovaný na tolik termů, kolik je jeho arita. Jinými slovy, pro každý predikátový symbol $P \in \mathbf{Pred}$ arity n a pro každou n-tici termů $t_1, t_2, ..., t_n$ je $P(t_1, t_2, ..., t_n)$ atomická formule.

3.1.5 Formule

Množina formulí je definována těmito pravidly:

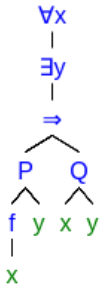
1. Každá atomická formule je formule.
2. Jsou-li φ a ψ dvě formule, pak $(\neg\varphi), (\varphi \wedge \psi), (\varphi \vee \psi), \varphi \Rightarrow \psi, (\varphi \Leftrightarrow \psi)$ jsou opět formule.
3. Je-li φ formule a x proměnná, pak $(\forall x\varphi)$ a $(\exists x\varphi)$ jsou opět formule.
4. Nic, co nevzniklo pomocí konečně mnoha použití bodů 1 až 3, není formule.

3.1.6 Derivační strom formule

To, jak vznikla formule skládající se z atomických formulí a ty z termů si můžeme znázornit na derivačním stromu dané formule. Jedná se o kořenový strom, kde každý vrchol, který není listem je ohodnocen logickou spojkou, kvantifikátorem, predikátovým symbolem nebo funkčním symbolem. Počet následovníků je u binárních logických spojek 2, u logických unárních spojek a kvantifikátorů 1 a u termů a funkcí je počet následovníků roven aritě. Listy stromu jsou ohodnoceny proměnnou nebo konstantou.

3.1.7 Volný a vázaný výskyt proměnné

Máme formuli φ a její derivační strom. List derivačního stromu obsazený proměnnou x je výskyt proměnné x ve formuli φ. Výskyt proměnné x je *vázaný* ve formuli φ, jestliže při postupu od listu ohodnoceného tímto x ve směru ke kořeni derivačního stromu narazíme na kvantifikátor s touto proměnnou. V opačném případě mluvíme o *volném* výskytu proměnné x.



Obrázek 3.1: Příklad derivačního stromu $\beta = \forall x \exists y(P(f(x), y) \Rightarrow Q(a, y))$

3.1.8 Sentence

Formule, která má pouze vázané výskyty proměnné, se nazývá *sentence*. Formuli, která má pouze volné výskyty proměnné, se říká *otevřená formule*.

3.1.9 Literál

Literál je atomická formule (tzv. pozitivní literál), nebo negace atomické formule (tzv. negativní literál). Komplementární literály jsou dva literály, z nichž jeden pozitivní, jeden je negativní a ten negativní je negací pozitivního.

3.1.10 Klauzule

Klauzule je sentence taková, že všechny kvantifikátory jsou obecné, stojí na začátku sentence (na jejich pořadí nezáleží) a za nimi následují literál nebo disjunkce konečně mnoha literálů.
Za klauzuli budeme považovat ještě formuli F zastupující kontradikci, kterou budeme nazývat prázdná klauzule.

3.1.11 Klauzární tvar

Pro každou sentenci φ existuje množina klauzulí S_φ taková, že sentence φ je splnitelná právě tehdy, když je splnitelná množina S_φ.
Jinými slovy se jedná vlastně o obdobu CNF z výrokové logiky.

3.1.12 Resolventa klauzulí

Máme dvě klauzule K_1 a K_2. Předpokládejme, že K_1 obsahuje literál L_1 a K_2 obsahuje literál L_2 pro něž existuje substituce θ taková, že $\theta(L_1)$ a $\theta(L_2)$ tvoří dvojici komplementárních literálů.
Pak resolventa klauzulí K_1 a K_2 je klauzule K, která je určena všemi literály obsaženými v $\theta(K_1) \setminus \theta(L_1)$ a $\theta(K_2) \setminus \theta(L_2)$; (tj. literály doplněné o obecné kvantifikátory všech proměnných, které se nacházejí v $\theta(K_1) \setminus \theta(L_1)$ a $\theta(K_2) \setminus \theta(L_2)$).

Neformálně řečeno, resolventu K dostaneme tak, že v "těle" klauzule necháme všechny literály z v $\theta(K_1) \setminus \theta(L_1)$ a v $\theta(K_2) \setminus \theta(L_2)$ a na začátek doplníme obecné kvantifikátory se všemi proměnnými, které v klauzuli zbyly.

3.1.13 Legální přejmenování proměnné

Přejmenování výskytů proměnné x ve formuli φ je legálním přejmenováním proměnné, jestliže

- Jedná se o výskyt vázané proměnné ve φ.
- Přejmenováváme všechny výskyty x vázané daným kvantifikátorem
- Po přejmenování se žádný dříve volný výskyt proměnné nesmí stát vázaným výskytem.

3.2 Sémantika logiky prvního řádu

3.2.1 Interpretace jazyka logiky prvního řádu

Interpretace logiky prvního řádu s predikátovými symboly **Pred**, konstantními symboly **Kons** a funkčními symboly **Func** je dvojice $\langle U, [\![-]\!] \rangle$, kde:

- U je neprázdná množina nazývaná *universum*.
- $[\![-]\!]$ je přiřazení, které:
  1. Každému predikátovému symbolu $P \in$ **Pred** arity n přiřazuje podmnožinu $[\![-]\!]$ množiny U^n, tj. n-ární relaci na množině U.
  2. Každému konstantnímu symbolu $a \in$ **Kons** přiřazuje prvek z U, značíme jej $[\![a]\!]$.
  3. Každému funkčnímu symbolu $f \in$ **Func** arity n přiřazuje zobrazení množiny U^n do U, značíme je $[\![f]\!]$.

  Množina U se někdy nazývá domain a označuje se D.

3.2.2 Kontext proměnných

Je dána interpretace $\langle U, [\![-]\!] \rangle$. *Kontext proměnných* je zobrazení ρ, které každé proměnné $x \in$ **Var** přiřadí prvek $\rho(x) \in U$. Je-li ρ kontext proměnných, $x \in$ **Var** a $d \in U$, pak

$$\rho[x := d]$$

označuje kontext proměnných, který má stejné hodnoty jako ρ pouze v proměnné x má hodnotu d. Kontextu proměnných $\rho[x := d]$ říkáme *update* kontextu ρ o hodnotu d v x.

3.2.3 Interpretace termů při daném kontextu proměnných

Je dána interpretace $\langle U, [\![-]\!] \rangle$ a kontext proměnných ρ. Pak termy interpretujeme následujícím způsobem.

1. Je-li term konstantní symbol $a \in \mathbf{Kons}$, pak jeho hodnota je prvek $[\![a]\!]_\rho = [\![a]\!]$. Je-li term proměnná x, pak jeho hodnota je $[\![x]\!]\rho = \rho(x)$.

2. Je-li $f(t1, ..., t_n)$ term. pak jeho hodnota je $[\![f]\!]_\rho = [\![f]\!]([\![t_1]\!]_\rho, ..., , [\![t_n]\!]_\rho)$.

3.2.4 Pravdivostní hodnota formule

Pravdivostní hodnota formule v dané interpretaci $\langle U, [\![-]\!] \rangle$ a daném kontextu ρ

1. Nechť φ je atomická formule. Tj. $\varphi = P(t_1, ..., t_n)$, kde P je predikátový symbol arity n a $t1, ..., t_n$ jsou termy. Pak φ je pravdivá v interpretaci $\langle U, [\![-]\!] \rangle$ a kontextu ρ právě tehdy, když $([\![t_1]\!]_\rho, ..., [\![t_n]\!]_\rho) \in [\![P]\!]$.

2. Jsou-li ρ a ψ formule, jejichž pravdivost v interpretaci $\langle U, [\![-]\!] \rangle$ a kontextu ρ již známe, pak:

   - $\neg\varphi$ je pravdivá právě tehdy, když ρ není pravdivá.
   - $\varphi \wedge \psi$ je pravdivá právě tehdy, když ρ i ψ jsou pravdivé.
   - $\varphi \vee \psi$ je nepravdivá právě tehdy, když ρ i ψ jsou nepravdivé.
   - $\varphi \Rightarrow \psi$ je nepravdivá právě tehdy, když ρ je pravdivá a ψ je nepravdivá.
   - $\varphi \Leftrightarrow \psi$ je pravdivá právě tehdy, když buď obě formule ρ a ψ jsou pravdivé, nebo obě formule ρ a ψ jsou nepravdivé.

3. Je-li φ formule a x proměnná, pak:

   - $\forall\varphi(x)$ je pravdivá právě tehdy, když formule φ je pravdivá v *každém* kontextu $\rho[x := d]$, kde d je prvek U.
   - $\exists\varphi(x)$ je pravdivá právě tehdy, když formule φ je pravdivá v *alespoň jednom* kontextu $\rho[x := d]$, kde d je prvek U.

3.2.5 Pravdivostní hodnota sentence

Sentence φ je pravdivá v interpretaci $\langle U, [\![-]\!] \rangle$ právě tehdy, když je pravdivá v každém kontextu proměnných ρ.

3.2.6 Model sentence

Interpretace $\langle U, [\![-]\!] \rangle$, ve které je sentence φ pravdivá, se nazývá *model sentence* φ.

3.2.7 Tautologie, kontradikce, splnitelná sentence

Sentence ϕ se nazývá *tautologie*, jestliže je pravdivá v každé interpretaci. Sentence se nazývá *kontradikce*, jestliže je nepravdivá v každé interpretaci. Nazývá se *splnitelná*, jestliže je pravdivá v aspoň jedné interpretaci.

3.2.8 Splnitelné množiny sentencí

Množina sentencí M je *splnitelná* právě tehdy, když existuje interpretace $\langle U, [\![-]\!] \rangle$, v níž jsou všechny sentence z M pravdivé. Takové interpretaci pak říkáme *model* množiny sentencí M. Množina sentencí M je *nesplnitelná*, jestliže ke každé interpretaci $\langle U, [\![-]\!] \rangle$ existuje formule z M, která je v $\langle U, [\![-]\!] \rangle$ nepravdivá. Z poslední definice vyplývá, že prázdná množina sentencí je splnitelná.

3.3 Sémantický důsledek (subsumpce)

3.3.1 Definice sémnatického důsledku

Řekneme, že sentence φ je sémantickým důsledkem, též konsekventem množiny sentencí S právě tehdy, když každý model množiny S je také modelem sentence φ. Tento fakt značíme

$S \models \varphi$

Můžeme též říci, že sentence φ není konsekventem množiny sentencí S, jestliže existuje model množiny S, který není modelem sentence φ. To znamená, že existuje interpretace $\langle U, [\![-]\!] \rangle$, v níž je pravdivá každá sentence z množiny S a není pravdivá formule φ.

3.3.2 Konvence

Jestliže množina sentencí S je jednoprvková, tj. $S = \{\psi\}$, pak píšeme $\psi \models \varphi$ místo $\{\psi\} \models \varphi$. Je-li množina S prázdná, píšeme $\models \varphi$ místo $\varphi \models \emptyset$.

3.3.3 Základní vztahy

1. Jeli $\varphi \in M$, je $M \models \varphi$
2. Je-li $N \subseteq M$ a $N \models \varphi$, je i $M \models \varphi$.
3. Je-li φ tautologie, pak $M \models \varphi$ pro každou množinu sentencí M.
4. Je-li $\models \varphi$, pak φ je tautologie.
5. Je-li M nesplnitelná množina, pak $M \models \varphi$ pro každou sentenci φ.
6. $\psi \models \varphi$ je splnitelná právě tehdy, když $\varphi \vee \neg\psi$ není splnitelná

3.4 Základní vztahy logiky prvního řádu

α, β a γ jsou formule, x je proměnná a P a Q jsou predikátové symboly

1. $\alpha \Rightarrow \beta \models\!\models \neg\alpha \vee \beta$
2. $\alpha \Leftrightarrow \beta \models\!\models (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$
3. $\neg\forall x\alpha \models\!\models \exists x\neg\alpha$
4. $\neg\exists x\alpha \models\!\models \forall x\neg\alpha$
5. $\neg(\alpha \wedge \beta) \models\!\models \neg\alpha \vee \text{`}\beta$
6. $\neg(\alpha \vee \beta) \models\!\models \neg\alpha \wedge \beta$
7. $\neg\neg\alpha \models\!\models \alpha$
8. $\alpha \wedge (\beta \wedge \gamma) \models\!\models (\alpha \wedge \beta) \wedge \gamma$
9. $\alpha \vee (\beta \vee \gamma) \models\!\models (\alpha \vee \beta) \vee \gamma$
10. $\alpha \wedge (\beta \vee \gamma) \models\!\models (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
11. $\alpha \vee (\beta \wedge \gamma) \models\!\models (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
12. $\alpha \wedge (\forall x\beta) \models\!\models \forall x(\alpha \wedge \beta)$
13. $\alpha \vee (\forall x\beta) \models\!\models \forall x(\alpha \vee \beta)$
14. $\alpha \wedge (\exists x\beta) \models\!\models \exists x(\alpha \wedge \beta)$
15. $\alpha \vee (\exists x\beta) \models\!\models \exists x(\alpha \vee \beta)$
16. $\forall x(\alpha \wedge \beta) \models\!\models (\forall x\alpha) \wedge (\forall x\beta)$
17. $\forall x(\alpha \vee \beta) \models\!\models (\forall x\alpha) \vee (\forall x\beta)$
18. $\exists x(\alpha \wedge \beta) \models\!\models (\exists x\alpha) \wedge (\exists x\beta)$
19. $\exists x(\alpha \vee \beta) \models\!\models (\exists x\alpha) \vee (\exists x\beta)$

3.5 Normálové formy

3.5.1 Negativní normální forma

Formule je v negativní normální formě, pokud jsou negace aplikovány pouze na atomické formule.

Příklad: $\forall x\neg P(x) \vee \neg Q(x)$

3.5.2 Prenexová normální forma

Formule je v prenexní normální formě, pokud je ve tvaru $Q_1x_1Q_2x_2....Q_nx_n\varphi$, kde $Q_i \in$ ($\forall$ nebo $\exists$), x_i je proměnná a φ je formule bez kvantifikátorů.

To znamená, že všechny kvantifikátory jsou před formulí a zbytek formule je bez kvantifikátorů.

Příklad: $\forall x \exists y(P(x, y) \vee Q(x, y))$

3.5.3 Skolemova normální forma

Formule je ve skolemově normální formě, pokud neobsahuje existenční kvantifikátor.

Příklad: $\forall x(P(x, f(x)) \vee Q(x, f(x)))$

3.5.4 Konjuktivní normální forma

Formule je v konjunktivní normální formě, pokud je ve tvaru konjunkce formulí, kde každá formule je disjunkce atomických formulí.

Příklad: $\forall x((\neg P(x, f(x)) \vee Q(x, f(x))) \wedge (P(x, f(x)) \vee \neg Q(x, f(x))))$

Kapitola 4

Logika prvního řádu s rovností

Definice pojmů logiky prvního řádu s rovností a jejich vysvětleni je citováno z [14]

Logika prvního řádu s rovností je rozšíření logiky prvního řádu bez rovnosti.

V této kapitole zavedeme nový logický symbol a to rovnost $=$ (nerovnost $\neg =$), upravíme definici atomické formule a přidáme nové základní vztahy a pojmy.

Logický symbol rovnost na rozdíl od ostatních logických symbolů lze aplikovat pouze na termy (na rozdíl od ostatních logických symbolů, které lze aplikovat pouze na predikáty.

4.1 Atomická formule

V logice prvního řádu s rovností je oproti logice prvního řádu bez rovnosti považován za atomickou formuli i výraz

$t_1 = t_2$ a $t_1 \neg = t_2$

kde t_1 a t_2 jsou termy.

4.2 Základní vztahy

x, $x_1, ..., x_n$ a y, $y_1, ..., y_n$ jsou proměnný.

1. Reflexibilita: $x = x$
2. Symetrie: $x = y \models\!\models y = x$
3. Trazitivita: $x = y \wedge y = z \models\!\models x = z$
4. Substituce funkcí:
   $x_1 = y_1 \wedge ... \wedge x_n = y_n \models\!\models f(x_1, ..., x_n) = f(y_1, ..., y_n)$, pro každý funkční symbol f
5. Substituce predikátu:
   $x_1 = y_1 \wedge ... \wedge x_n = y_n \wedge P(x_1, ..., x_n) \models\!\models P(y_1, ..., y_n)$ pro každý predikátový symbol P.

4.3 Mělký literál

Literál je mělký (shallow), právě když je v jednom z následujících tvarů:

1. $P(x_1, ..., x_n)$ nebo $\neg P(x_1, ..., x_n)$
2. $f(x_1, ..., x_n) = y$ nebo $f(x_1, ..., x_n)\neg = y$
3. $x = y$

Kde:
x, y a $x_1, ..., x_n$ jsou proměnné
f je funkční symbol
P je predikátový symbol

4.4 Plochá klauzule

Plochá klauzule je taková klauzule, která obsahuje pouze mělký literály.

4.5 Pramodulation

Definice přepisovacího pravidla pro rovnost.

- C_1: $L[t] \vee C_1^{'}$
- C_2: $r = s \vee C_2^{'}$
- σ je MGU for t a r

Klauzule v tomto tvaru můžeme přepsat na $L\sigma[s\sigma] \vee C_1^{'}\sigma \vee C_2^{'}\sigma$
kde $C[t]$ značí, že C obsahuje term t a $C\sigma$ značí aplikování unifikace.

Kapitola 5

Analýza

5.1 Knihovna SymPy

Podle zadání má implementace algoritmů organicky doplňovat knihovnu SymPy. Z tohoto důvodu se u části analýzy omezím pouze na rozbor toho, jak je to řešené v této knihovně. Nemá totiž smysl zvažovat řešení, které se liší od způsobu jak je to řešeno ve zbytku knihovny.

Využití jiného řešení například pro testování by totiž znamenalo, že by práce nezapadala do knihovny SymPy, nebo by znamenala změnu testování v celé knihovně. To není tématem této práce. Takto velká změna v rozsáhlém projektu není pouze otázka, jestli je dané řešení lepší, ale hlavně otázka diskuze a konzultace s ostatními přispěvateli do projektu. Protože záleží i na jejich preferencích/znalostech a jestli výhody převýší problémy způsobené touto změnou.

5.1.1 Testování

Knihovna SymPy má vlastní testovací framework, který vychází z py.test[1]. Tento testovací framework má podle dokumentace fungovat velmi podobně nebo stejně jako py.test. Toto řešení je hlavně z důvodu snahy udržet knihovnu SymPy bez jakýchkoliv externích závislostí [2].

Testy jsou rozděleny na 2 druhy.

- Za prvé to jsou klasické testy, rozdělené do jednotlivých souborů, které obsahují jenom testy, například:

  dummy_ test_ file.py

  ```
  def test_dummyTest():
      assert 1+1 == 2
  def test_anotherDummyTest()
      a = 1
  ```

[1] py.test testovací framework <`https://pytest.org`>
[2] Toto tvrzení vychází z dokumentace frameworku.

```
b = 2
assert b - a == a
```

- Za druhé se jedná o doctesty. Testovací framework projde texty v docstring[5] a hledá kusy textu, které odpovídají formátu Python console (jsou uvozeny >>>). Ty poté je zkusí spustit a zkontroluje jestli výsledek odpovídá uvedenému výsledku v dokumentaci.

  Tenhle typ testů se hodí pro základní jednoduché testy a zároveň testuje, jestli úkázky kódu v dokumentaci jsou stále aktuální. Příklad definice funkce *squareSum* s docstring obsahují doctesty:

```
def squareSum(a, b):
    """Function squareSum returnsum of squares. That mean a^2 + b^2
    example:
    >>> squareSum(1,2)
    5
    >>> squareSum(-2,2)
    8
    """
    return a**2 + b**2
```

  V tomto příkladu bude při testování zavolána funkce *squreSum* s parametry $1, 2$ a očekáván výsledek 5. Poté bude zavolána tato funkce s parametry $-2, 2$ a je očekáván výsledek 8.

5.1.2 Dokumentace

Dokumentace je automaticky generována z docstring. Většina funkcí/metod má v kódu obsáhlou dokumentaci v docstringu i s několika ukázky použití. Dokumentaci lze vygenerovat na linuxu pomocí příkazu *"make html"*

5.1.3 Knihovny

SymPy se snaží o nezávislost na ostatních knihovnách a je čistě napsána v Pythonu. Z toho důvodu má knihovna pouze jednu závislost na externí knihovně a tou je knihovna mpmath[3].

Dále má několik volitelných závislostí - nejsou vyžadovány pro fungování knihovny. Příklad volitelné závislosti je knihovna gmpy[4]. Pokud je tato knihovna dostupná, tak modul polynomial využívá její datové typy a pro zrychlení výpočtů. Knihovna gmpy je napsaná v C++ a lze s ní dosáhnout zrychlení výpočtů o několik řádů.

[3] mpmath knihovna <http://mpmath.org/>

[4] gmpy knihovna <https://code.google.com/archive/p/gmpy/>

5.2 Algoritmy

Všechny algoritmy předpokládají na vstupu sentenci ve formátu derivačních stromů. Pokud by algoritmus předpokládal jiný vstup tak na to bude upozorněno. Všechny algoritmy také předpokládají, že proměnné jsou unikátní. Unikátností je myšleno, že se proměnná vyskytuje pouze s jedním kvantifikátorem. Pro normalizaci názvů proměnných je k dispozici první algoritmus 5.2.1

5.2.1 Normalizace proměnných

Přejmenování proměnných se stejným názvem, ale v různých kontextech tak, aby se proměnná vyskytovala pouze ve svém kontextu (pouze v podstromu daného kvantifikátoru)

5.2.1.1 Algoritmus

1. Algoritmus prochází derivační strom formule do hloubky.
2. Pokud narazí na uzel ohodnocený kvantifikátorem zkontroluje, jestli proměnná nebyla už někde jinde použita.
   (a) Pokud proměnná ještě nebyla nikde použita, je označena jako použitá a algoritmus pokračuje dál.
   (b) Pokud proměnná je označená jako použitá tak ji algoritmus přejmenuje tak, aby měla unikátní název. Zároveň přejmenuje tuto proměnnou v celém podstromu tohoto kvantifikátoru a pokračuje dál.

5.2.1.2 Pseudokód

```
1: procedure NormalizeVariableName(formula)
2:     NormalizeVariableName(formula, new HashMap, 0)
3: end procedure
4:
5: procedure NormalizeVariableNames(formula, usedVariables, newVariablesIndex)
6:     renamedVariable ← false
7:
8:     if formula.isQuantificator() then
9:         varName ← formula.getVariableName()
10:        if usedVariables.contain(varName) then
11:            renamedVariable ← true
12:            usedVariables[varName] ← "v" + newVariablesIndex
13:            newVariablesIndes + +
14:        else
15:            usedVariables[varName] = varName
16:        end if
17:    end if
18:
```

```
19:     if formula.isVariable then
20:         formula.renameVariable(usedVariables[formula.getVariableName()])
21:         return
22:     end if
23:
24:     for child in formula.getChilds() do
25:         NormalizeVariableNames(child, usedVariables)
26:     end for
27:
28:     if renamedVariable then
29:         varName ← formula.getVariableName()
30:         usedVariables[varName] = varName
31:     end if
32: end procedure
```

5.2.1.3 Konečnost algoritmu

- Protože formule je konečné délky, tak i její derivační strom je konečný.
- Algoritmus prochází derivační strom formule jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).
- V každém kroku algoritmus zkontroluje jestli, je daná proměnná již použitá což vhledem ke konečnému počtu proměnných lze provést v konečném počtu kroků.

Algoritmus je konečný

5.2.1.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Derivační strom formule je procházen do hloubky, což je n kroků.
- V nejhorším případě je v každém kroku kontrolováno, jestli proměnná existuje nebo je proměnná označena jako použitá (uložení proměnné do slovníku). To znamená, že vyhledání ve slovníku nebo zápis do slovníku má časovou složitost obou kroků $\Theta(n)$ (amortizovaná složitost těchto operací je $\Theta(1)$). V časové složitosti můžeme použít n vztahující se k délce formule, protože složitost těchto operací je sice vztažena k počtu prvků ve slovníku. Počet prvků ve slovníku se v nejhorším případě asymptoticky blíží k počtu uzlů.
- Ostatní operace v každém kroku jsou s konstantní složitostí.

Celkově tedy provedeme n kroků na projití formule a v každém kroku nejhůře n operací.
Časová složitost je: $\Theta(n^2)$
Amortizovaná časová složitost: $\Theta(n)$

5.2.2 Převedení do negativní normální formy

5.2.2.1 Algoritmus

1. Pomocí vztahů 1-2 z 3.4 odstraníme z formule implikaci a ekvivalenci.
   (a) Algoritmus prochází strom formule do hloubky.
   (b) Pokud narazí na uzel obsahující implikaci nebo ekvivalenci tak tento uzel a jeho podstrom upraví podle odpovídajícího vztahu. Poté pokračuje v procházení do hloubky nového stromu.
2. Dále pomocí vztahů 3-7 z 3.4 přesune negaci před atomické formule.
   (a) Algoritmus prochází strom formule do hloubky.
   (b) Pokud narazí na uzel obsahující negaci, která není před atomickou formulí, tak tento uzel a jeho podstrom upraví podle odpovídajícího vztahu. Poté pokračuje v procházení do hloubky nového stromu.

5.2.2.2 Pseudokód

```
1: procedure RemoveImplication(formula)
2:     if formala.isImplication() then
3:         notNode ← newNotNode()
4:         notNode.setChild(formula.getLeftNode())
5:         disjunctionNode ← newDisjunctionNode()
6:         disjunctionNode.setLeftChild(notNode)
7:         disjunctionNode.setRightChild(formula.getRightChild())
8:
9:         formula.replaceWithNewNode(newdisjunctionNode)
10:
11:        for child in formula.getChilds() do
12:            BubleNegation(child)
13:        end for
14:    end if
15: end procedure
16:
17: procedure RemoveEquivalence(formula)
18:    if formala.isEquivalence() then
19:        notNode1 ← newNotNode()
20:        notNode1.setChild(formula.getLeftNode())
21:        disjunctionNode1 ← newDisjunctionNode()
22:        disjunctionNode1.setLeftChild(notNode1)
23:        disjunctionNode1.setRightChild(formula.getRightChild())
24:
25:        notNode2 ← newNotNode()
26:        notNode2.setChild(formula.getRightNode())
27:        disjunctionNode2 ← newDisjunctionNode()
```

```
28:         disjunctionNode2.setLeftChild(formula.getLeftChild())
29:         disjunctionNode2.setRightChild(notNode1)
30:
31:         conjuctionNode ← newConjuctionNode()
32:         conjuctionNode.setLeftChild(disjunctionNode1)
33:         conjuctionNode.setRightChild(disjunctionNode2)
34:
35:         formula.replaceWithNewNode(newconjuctionNode)
36:
37:         for child in formula.getChilds() do
38:             BubleNegation(child)
39:         end for
40:     end if
41: end procedure
42:
43: ... Similar procedures for rules 3-7
44:
45: procedure BubleNegation(formula)
46:     if formula.isNegation() then
47:         childNode ← formula.getChild()
48:         if childNode.isAtomicFormula() then
49:             return
50:         else if childNode.isForAll() then
51:             rule3(formula)
52:         else if childNode.isExists() then
53:             rule4(formula)
54:         else if .... then
55:             ...Similar conditions for other rules.
56:         end if
57:     end if
58:     for child in formula.getChilds() do
59:         BubleNegation(child)
60:     end for
61: end procedure
62:
63: procedure ToNegationNormalForm(formula)
64:     RemoveImplication(formula)
65:     RemoveEquivalence(formula)
66:     BubleNegation(formula)
67: end procedure
```

5.2.2.3 Konečnost algoritmu

- Protože formule je konečné délky tak i její derivační strom je konečný.
- Odstranění implikací

  - Algoritmus jednou projde derivační strom formule.
  - V každém kroku zkontroluje jestli narazil na implikaci a pokud ano, tak ji podle pravidla pro implikaci odstraní což je v konstantním počtu operací.

  Odstranění implikací je konečné.

- Odstranění ekvivalencí

  - Algoritmus prochází derivační strom formule jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).
  - V každém kroku zkontroluje jestli narazil na ekvivalenci a pokud ano, tak ji podle pravidla pro ekvivalence odstraní což je v konstantním počtu operací.

  Odstranění ekvivalencí je konečné.

- Přesun negací před atomické formule

  - Algoritmus jednou projde derivační strom formule.
  - V každém kroku zkontroluje jestli narazil na negaci, jejíž potomek není atomická formule. Pokud potomek není atomická formule, tak ji podle jednoho z pravidel přesune o úroveň níž, což je v konstantním počtu operací.
  - Vzhledem k tomu, že strom je konečné délky, tak i jeho hloubka musí být konečná. Z tohoto důvodu lze přesunout negaci o úroveň níž v konečném počtu kroků.

  Přesun negací před atomické formule konečné.

Protože všechny 3 nezávislé části jsou konečné tak i celý algoritmus je konečný.
Algoritmus je konečný

5.2.2.4 Časová složitost

Časová složitost je vztažena k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Odstranění implikací

  - Derivační strom formule je procházen do hloubky. Při proházení do hloubky vykoná algoritmus n kroků.
  - V nejhorším případě je v každém kroku odstraněna implikace. Odstranění implikace je v konstantním počtu operací.

  V celkově tedy provedeme $\Theta(n)$ operací.

- Odstranění ekvivalencí

  - Derivační strom formule je procházen do hloubky. Při proházení do hloubky vykoná algoritmus n kroků.

  - V nejhorším případě je v každém kroku odstraněna ekvivalence. Odstranění ekvivalence je v konstantním počtu operací.

  V celkově tedy provedeme $\Theta(n)$ operací.

- Přesun negací před atomické formule

  - Derivační strom formule je procházen do hloubky. Při proházení do hloubky vykoná algoritmus n kroků.
  - V nejhorším případě je v každém kroku přesuneme podle pravidel negaci blíže k atomické formuli. Přesun negace podle pravidel pro negaci je v konstantním počtu operací.
    Při přesunu negace do hloubky se velikost derivačního stromu nezmění.

  V celkově tedy provedeme $\Theta(n)$ operací.

Celý algoritmus provede $3 * n$ operací. **Časová složitost je:** $\Theta(n)$

5.2.3 Převod do prenexní normální formy

5.2.3.1 Algoritmus

Algoritmus předpokládá na vstupu formuli v negativní normálové formě.

1. Algoritmus si na začátku inicializuje prázdný zásobník.
2. Algoritmus prochází strom formule do hloubky.
   (a) Pokud algoritmus narazí na kvantifikátor, tak si ho s proměnnou přidá do zásobníku a odebere ze stromu formule.
3. Po projití celého stromu, algoritmus vezme frontu a kvantifikátory v pořadí, v jakém je přidával do zásobníku, je přidá zpátky na začátek formule (formule je před tímto krokem bez kvantifikátorů).

5.2.3.2 Pseudokód

```
1: procedure RemoveAllQuantifiers(formula, stack)
2:     if formula.isQuantifiers() then
3:         stack.put(newQuantifier(formula.getQuantifier(), formula.getVariable()))
4:         formula.replaceWithNewNode(formula.getChild())
5:     end if
6:
7:     for child in formula.getChilds() do
8:         RemoveAllQuantifiers(child, stack)
9:     end for
10: end procedure
11:
12: procedure AddQuantifiersFromStack(formula, stack)
```

```
13:     for quantifier in stack.reverseOrder() do
14:         quantifier.setChild(formula)
15:         formula ← quantifier
16:     end for
17: end procedure
18:
19: procedure ToPrenexNormalForm(formula)
20:     stack ← newStack()
21:     RemoveAllQuantifiers(formula, stack)
22:     AddQuantifiersFromStack(formula, stack)
23: end procedure
```

5.2.3.3 Konečnost algoritmu

- Protože formule je konečné délky, tak i její derivační strom je konečný.
- Odstraněni kvantifikátorů
  - Algoritmus prochází derivační strom formule jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).
  - V každém kroku zkontroluje jestli narazil na kvantifikátor a pokud ano tak ho odebere ze stromu a vloží do zásobníku. Oba kroky jsou v konstantním počtu operací.

  Odstranění kvantifikátorů je konečné.
- Vložení kvantifikátorů zpátky do formule
  - Algoritmus postupně odebírá kvantifikátory ze zásobníku dokud není prázdný. Protože zásobník má velikost rovnou počtu proměnných, kterých je konečný počet, tak i postupné vyprázdnění zásobníku je v konečném počtu kroků (odebrání proměnné ze zásobníku je v konstantním počtu operací).
  - V každém kroku vytvoří nový kvantifikátor a zařadí ho na začátek formule. Tato operace je v konstantním počtu operací.

  Vložení kvantifikátorů zpátky do formule je konečné.

Protože obě nezávislé části jsou konečné tak i celý algoritmus je konečný.
Algoritmus je konečný

5.2.3.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Odstranění kvantifikátorů

  - Derivační strom formule je procházen do hloubky. Při proházení do hloubky vykoná algoritmus n kroků.
  - V každém kroku algoritmus zkontroluje jestli narazil na kvantifikátor. Pokud narazil na kvantifikátor, tak ho odebere ze stromu a vloží do zásobníku. Odebrání kvantifikátoru, ze stromové struktury je v konstantním počtu operací. Jeho přidání do zásobníku je také v konstantním počtu operací.

  V celkově algoritmus provede $\Theta(n)$ operací.

- Přidání kvantifikátorů zpátky do formule

  - Algoritmus postupně odebírá kvantifikátory ze zásobníku dokud není zásobník prázdný. Protože zásobník má velikost rovnou počtu proměnných, který se v nejhorším případě asymptoticky blíží k délce formule, algoritmus provede n kroků.
  - V každém kroku algoritmus tak vytvoří nový kvantifikátor a zařadí ho na začátek formule, což je v konstantním počtu operací.

  V celkově algoritmus provede $\Theta(n)$ operací.

Celková složitost algoritmu je $2 * n$ operací.
Časová složitost je: $\Theta(n)$

5.2.4 Převod do skolemovi normální formy

Princip převodu do skolemovi normální formy (odstranění existenčních kvantifikátorů) je nahrazení proměnných, které se vážou k existenčnímu kvantifikátoru skolemovou funkcí. Jedná se o takovou funkci, která má za parametry všechny proměnné, které se vážou k obecným kvantifikátorům a jsou v derivačním stromu nad danným existenčním kvantifikátorem.

Jinak řečeno, pokud formule říká, že pro všechny x existuje y, tak můžeme y nahradit funkcí, která pro dané x vrací takové y pro které daný výraz platí. V případě, že funkce by neměla žádné parametry, je funckce nahrazena konstantou.
Například:

$\forall x \exists y P(x, y)$

můžeme přepsat na

$\forall x P(x, f(x))$

5.2.4.1 Algoritmus

Algoritmus předpokládá na vstupu formuli v negativní normální formě.

1. Algoritmus si na začátku inicializuje prázdný zásobník a slovník.
2. Algoritmus prochází strom formule do hloubky.

(a) Pokud algoritmus narazí na obecný kvantifikátor, tak si proměnnou přidá do zásobníku a pokračuje dál v procházení do hloubky. Při vracení se ve stromové struktuře zpátky ke kořenu, algoritmus na stejném místě proměnnou odebere ze zásobníku.

(b) Pokud algoritmus narazí na existenční kvantifikátor, tak si vytvoří novou skolemovu funkci a tu přidá do slovníku a jako klíč použije název proměnné. Nová skolemova funkce má jako parametry všechny proměnné ze zásobníku.

(c) Pokud algoritmus narazí na proměnnou, tak se podívá do slovníku, jestli v něm existuje klíč s názvem dané proměnné. Pokud existuje, tak proměnnou nahradí odpovídající skolemovou funkcí.

5.2.4.2 Pseudokód

```
1: procedure ToSkolemNormalForm(formula)
2:     ToSkolemNormalForm(formula, newStack(), newDictonary())
3: end procedure
4:
5: procedure ToSkolemNormalForm(formula, variables, skolemFunctions)
6:     if formula.isForAllQuantifier() then
7:         variables.put(formula.getVariableName())
8:     else if formula.isExistsQuantifier() then
9:         if variables.len == 0 then
10:            skolemFunctions[formula.getVariableName()] ← newConstant()
11:        else
12:            skolemFunctions[formula.getVariableName()] ← newSkolemFunction(variables)
13:        end if
14:    else if formula.isVariable() and skolemFunctions.hasKey(formula.getVariableName())
   then
15:        formula.replaceWithNewNode(skolemFunctions[formula.getVariableName()])
16:    end if
17:
18:    for child in formula.getChilds() do
19:        ToSkolemNormalForm(child, variables, skolemFunctions)
20:    end for
21:
22:    if formula.isForAllQuantifier() then
23:        variables.pop()
24:    end if
25: end procedure
```

5.2.4.3 Konečnost algoritmu

- Protože formule je konečné délky, tak i její derivační strom je konečný.
- Algoritmus prochází derivační strom formule jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).

- Algoritmus v každém kroku zkontroluje, jestli nenarazil na proměnnou nebo kvantifikátor.
  - Pokud algoritmus narazí na proměnnou, tak se podívá do slovníku jestli má proměnnou přepsat na skolemovu funkci. Vyhledání ve slovníku je v lineárním počtu operací.
  - Pokud algoritmus narazí na obecný kvantifikátor, tak proměnnou přidá do zásobníku, což je v konstantním počtu operací.
  - Pokud algoritmus narazí na existenční kvantifikátor, tak přidá novou skolemovu funkci do slovníku. Vytvoření skolemovi funkce a vložení do slovníku je v konečném počtu operací.

Algoritmus projde celý strom v konečném počtu kroků a v každém kroku provede konečný počet operací.
Algoritmus je konečný

5.2.4.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Algoritmus jednou projde derivační strom formule do hloubky, při tom provede n kroků.
- Algoritmus v každém kroku zkontroluje, jestli nenarazil na proměnnou nebo kvantifikátor.
  - Pokud algoritmus narazí na proměnnou, tak se podívá do slovníku jestli se má proměnná přepsat na odpovídající skolemovu funkci. Vyhledání ve slovníku je v čase $\Theta(n)$. Jako parametr časové složitosti zde můžeme použít n, které odpovídá délce formule. Je to z důvodu, že délka slovníku závisí na počtu proměnných a v nejhorším případě se počet proměnných asymptoticky blíží k počtu uzlů.
  - Pokud algoritmus narazí na obecný kvantifikátor, tak proměnnou přidá do zásobníku což je v konstantním počtu operací.
  - Pokud algoritmus narazí na existenční kvantifikátor, tak přidá novou skolemovu funkci do slovníku. Vytvoření skolemovi funkce je v čase $\Theta(n)$ (je potřeba projít celý zásobník, který může mít délku až n - viz předchozí bod). Vložení do slovníku je v čase $\Theta(n)$ (viz první bod).

Celkově tedy algoritmus provede n kroků a v každém kroku v nejhorším případě n operací.
Časová složitost je: $\Theta(n^2)$

5.2.5 Převod formule na klauzární tvar

5.2.5.1 Algoritmus

Algoritmus předpokládá na vstupu formuli v skolemově normální formě a zároveň prenexní normální formě.

1. První část algoritmu - přesun disjunkce ($\vee$) co nejníže v derivačním stromu, přesun konjunkce ($\wedge$) co nejvýše v derivačním stromu a odebrání kvantifikátorů.
   (a) Algoritmus prochází derivační strom formule do hloubky.
   (b) Algoritmus odebere všechny obecné kvantifikátory.
   (c) Pokud algoritmus narazí na disjunkci, zkontroluje jestli alespoň jeden potomek je konjunkce.
   Pokud ano, tak můžou nastat tyto možnosti:
      i. Levý potomek je konjunkce, pod-formule je tedy ve tvaru $(\alpha \wedge \beta) \vee \gamma$.
      V tomto případě prohodíme levého potomka za pravého, takže pod-formule je ve tvaru $\alpha \vee (\beta \wedge \gamma)$ a algoritmus pokračuje bodem (iii.).
      ii. Oba potomci jsou konjunkce, pod-formule je tedy ve tvaru $(\alpha \wedge \beta) \vee (\gamma \wedge \delta)$.
      V tomto případě můžeme celého levého potomka považovat za α, takže pod-formule je ve tvaru $\alpha \vee (\gamma \wedge \delta)$ a algoritmus pokračuje bodem (iii.).
      iii. Pravý potomek je konjunkce, pod-formule je tedy ve tvaru $\alpha \vee (\beta \wedge \gamma)$.
      Algoritmus aplikuje vztah $\alpha \vee (\beta \wedge \gamma) \models\!\models (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$ (viz 3.4) a pokračuje bodem a.
   (d) Pokud algoritmus nenarazil na žádnou disjunkci po které by v jednom z potomků následovala konjunkce, algoritmus pokračuje na část 2.
2. Druhá část algoritmu - převedení formule na množinu klauzulí.
   (a) Algoritmus prochází derivační strom formule do hloubky.
   (b) Pokud algoritmus narazí na konjunkci tak:
      i. Odebere konjunkci z derivačního stromu i s celým jejím podstromem.
      ii. Algoritmus aplikuje na oba potomky druhou část algoritmu a výsledné seznamy sloučí.
      iii. Pokud algoritmus nenalezne v derivačním stromě konjunkci, tak zpracovávaná formule je klauzulí a algoritmus ji přidá do seznamu (listu) klauzulí.
3. Třetí část - Přidání zpátky obecné kvantifikátory k klauzuli.
   (a) Algoritmus prochází derivační strom každé klauzule zvlášť do hloubky.
   (b) Pokud narazí na proměnnou, kterou nemá v seznamu proměnných, tak její název přidá do seznamu proměnných.
   (c) Po projití celého derivačního stromu klauzule, algoritmus na začátek derivačního stromu přidá obecné kvantifikátory pro všechny proměnné uložené v seznamu.

5.2.5.2 Pseudokód

```
1: function MoveDownDisjuction(formula)
2:     if formula.isQuantifier() then
3:         formula.replaceWithNewNode(formula.getChild())
4:         MoveDownDisjuction(formula)
5:         return true
```

```
    end if

    if formula.isDisjunction() then
        leftChild ← formula.getLeftChild()

        if leftChild.isConjuction() then
            formula.setLeftChild(formula.getRightChild())
            leftChild ← formula.getLeftChild()

            formula.setRightChild(leftChild)
        end if

        rightChild ← formula.getRightChild()
        if rightChild.isConjuction() then
            disjunction1 ← new Disjunction(leftChild, rightChild.getLeftChild())
            disjunction2 ← new Disjunction(leftChild, rightChild.getRightChild())
            conjuction ← new Conjunction(disjunction1, disjunction2)
            formula.replaceWithNewNode(conjuction)

            return true
        end if
    end if
    return false
end function

function CNFFormulaToListClauses(formula)
    if formula.disjunction() then
        list ← []
        list.add(CNFFormulaToListClauses(formula.getLeftChild()))
        list.add(CNFFormulaToListClauses(formula.getRightChild()))
        return list
    else
        return [formula]
    end if
end function

procedure GetAllVariables(clause)
    if clause.isVariable() then
        var ← clause.getVariable()
        return [var]
    end if

    vars ← []
    for child in clause.getChilds do
        vars.add(GetAllVariables(child))
    end for
```

```
52:     return vars
53: end procedure
54:
55: function FormulaToListClauses(formula)
56:     while MoveDownDisjuction(formula) do
57:     end while
58:
59:     clauses ← CNFFormulaToListClauses(formula)
60:
61:     for clause in clauses do
62:         vars ← AddQuantifiersToClause(clauses)
63:         clause.createUniversalQuantifies(vars)
64:     end for
65:
66:     return clauses
67: end function
```

5.2.5.3 Konečnost algoritmu

- První část algoritmu - přesun disjunkce ($\vee$) co nejníže v derivačním stromu a zároveň přesun konjunkce ($\wedge$) co nejvýše v derivačním stromu.

  1. Protože formule je konečné délky, tak i její derivační strom je konečný.
  2. Pokud algoritmus narazí na obecný kvantifikátor tak ho odebere, tuto operaci provede maximálně n krát.
  3. Pokud algoritmus narazí na disjunkci, tak zkontroluje jestli potomci neobsahují konjunkci. Tato operace je v konstantním počtu operací.
     Pokud nějaký potomek obsahuje konjunkci tak jí posune o úroveň níž (což je konstantní počet operací) a začne od začátku.
  4. Vzhledem k tomu, že strom je konečné hloubky a tak posun konjunkce o úroveň níž lze provést v pouze konečně-krát.

  Tato část je v konečném počtu kroků.

- Druhá část algoritmu - převedení formule na množinu klauzulí.

  1. Protože formule je konečné délky, tak i její derivační strom je konečný.
  2. V každém kroku se algoritmus buď zanoří níže v derivačním stromu, nebo rozdělí pod-formuli na dvě menší.
  3. Vzhledem k tomu, že počet konjunkcí v konečném derivačním stromu je konečný, tak počet rozdělení formule je také konečný.

  Tato část je v konečném počtu kroků.

- Třetí část - Přidání obecných kvantifikátorů na začátek klauzulí.

1. Počet klauzulí vzniklých je konečný počet (jedná se vlastně o rozdělení derivačního stromu původní formule na několik menších)
2. Protože klauzule je konečné délky, tak i její derivační strom je konečný.
3. Algoritmus prochází derivační strom klauzule jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).
4. Algoritmus v každém kroku zkontroluje, jestli nenarazil na proměnnou.
5. Pokud narazí na proměnnou, tak její název přidá do seznamu proměnných. Tento krok je konečném počtu operací.

Tato část je v konečném počtu kroků.

Vzhledem k tomu, že všechny části jsou v konečném počtu kroků, tak i celý algoritmu je konečný
Algoritmus je konečný.

5.2.5.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- První část algoritmu - přesun disjunkce ($\vee$) co nejníže v derivačním stromu, přesune konjunkce ($\wedge$) co nejvýše v derivačním stromu a odstraní kvantifikátory

  1. Algoritmus projde derivační strom formule do hloubky, při tom provede n kroků.
  2. Pokud algoritmus narazí na obecný kvantifikátor tak ho odebere, tento krok je v konstantním počtu operací.
  3. Pokud algoritmus narazí na disjunkci, tak zkontroluje jestli potomci neobsahují konjunkci. Tato operace je v konstantním počtu operací.
     Pokud nějaký potomek obsahuje konjunkci tak jí posune o úroveň níž (což je konstantní počet operací) a začne od začátku. Při této operaci se velikost stromu až zdvojnásobí.
  4. Vzhledem k tomu, že strom má hloubku maximálně n, tak posun konjunkce o úroveň výš může být proveden maximálně n-krát .

  Protože formule se zdvojnásobuje s každým posunem konjunkce výš, tak algoritmus v této části má časovou složitost $\Theta(2^n)$.

- Druhá část algoritmu - převedení formule na množinu klauzulí.

  1. Algoritmus projde derivační strom formule do hloubky, při tom provede n_1 kroků.
  2. Protože formule je konečné délky, tak i její derivační strom je konečný.
  3. V každém kroku se algoritmus buď zanoří níže v derivačním stromu, nebo rozdělí pod-formuli na dvě menší.
     Oba kroky jsou v konstantním počtu operací.

Algoritmus ve skutečnosti projde derivační strom jen jednou, protože při odebrání konjunkce a rozdělení stromu na dva menší, algoritmus akorát zavolá sám sebe na oba potomky konjunkce, které ještě neprošel. Protože velikost derivačního stromu je 2^n oproti původní délce ($n_1 = 2^n$) tak tato část má časovou složitost $\Theta(2^n)$.

- Třetí část - Přidání obecných kvantifikátorů ke klauzulím.

  1. Algoritmus projde derivační strom každé klauzule do hloubky.
     Celková délka všech klauzulí je 2^n, protože formule vzniklá po prvním kroku je akorát rozdělená na několik menších částí.
  2. Algoritmus v každém kroku zkontroluje, jestli nenarazil na proměnnou (konstantním počet operací).
  3. Pokud narazí na proměnnou, tak její název přidá do slovníku proměnných. Tento krok je v lineárním počtu operací.
     Maximální délka slovníku je ale pouze n (při transformaci stromu se počet proměnných nezměnil)
  4. Vytvoření nových kvantifikátorů závisí na počtu různých proměnných, v klauzuli. Každá klauzule může obsahovat maximálně n různých proměnných.
  5. Vytvoření kvantifikátoru je v konstantním počtu operací

  V této části algoritmus provede $2^n * n$ operací.
  Tato část má časovou složitost $\Theta(2^n)$.

Vzhledem k tomu, že všechny části jsou nezávislé, tak jejich časová složitost se rovná součtu jednotlivých časových složitostí.
Časová složitost je: $\Theta(2^n)$

5.2.5.5 Heuristika

Vzhledem k velké časové náročnosti algoritmu, je dobré se zamyslet nad heuristikou, která by urychlila výpočet. Bohužel zde je velká časová náročnost daná exponenciálním růstem formule při převodu do klauzárního tvaru a to nelze urychlit heuristikou (algoritmus musí postupně vytvořit celou novou formuli).

Heuristika neexistuje.

5.2.6 Unifikační algoritmus

Vstupem jsou 2 pozitivní literály L_1 a L_2, které nemají společné proměnné. Výstupem algoritmu je substituce pro unifikaci literálů, eventuálně chyba pokud taková substituce neexistuje. Popis algoritmu pracuje s literály, jako kdyby se jednalo o řetězec znaků.

5.2.6.1 Algoritmus

Popis algoritmu jsem převzal z *Matematická logika, Marie Demlová*[10]

1. Algoritmus si vytvoří prázdný list *sub*.

2. Jsou-li L_1, L_2 prázdné řetězce, algoritmus končí a hledaná substituce je v *sub*.

3. Označíme X_1 první znak řetězce L_1 a X_2 první znak řetězce Y_2.

4. Pokud se X_1 a X_2 rovnají tak je odstraníme z počátku L_1, L_2.
   Pokračujeme krokem 2.

5. Je-li X_1 proměnná, neděláme nic.
   Je-li X_2 proměnná, tak přehodíme L_1, L_2 a X_1, X_2.
   Není-li ani X_1 ani X_2 proměnná tak algoritmus končí a vrací chybu.

6. Je-li X_2 proměnná nebo konstanta, přidáme do *sub* substituci X_1 za X_2.
   X_1 a X_2 odstraníme z počátku L_1, L_2.
   Pokračujeme krokem 2.

7. Je-li X_2 funkční symbol, do X_2 dáme celý výraz skládající se ze všech jeho vstupů a závorek.
   Jestliže X_2 obsahuje X_1 algoritmus končí a vrací chybu. Jinak přidáme do *sub* substituci X_1 za X_2.
   X_1 a X_2 odstraníme z počátku L_1, L_2.
   Pokračujeme krokem 2.

5.2.6.2 Pseudokód

```
1:  procedure UnificationTerm(t1, t2, sub)
2:      if t1 == t2 then
3:          return sub
4:      end if
5:
6:      if not t1.isVariable() AND not t2.isVariable then
7:          return error
8:      end if
9:
10:     if t1.isFunction() AND t2.isFunction() then
11:         if t1.getFunctionSymbol() == t2.getFunctionSymbol() then
12:             t1Subterms = t1.getChilds()
13:             t2Subterms = t2.getChilds()
14:
15:             for i in t1Subterms.length do
16:                 sub ← UnificationTerm(t1SubTerms[i], t2SubTerms[i], sub)
17:                 if sub == error then
18:                     return error
19:                 end if
20:             end for
21:             return sub
22:         else
23:             return error
24:         end if
```

```
25:     end if
26:
27:     if not t1.isVariable() AND t2.isVariable then
28:         t3 ← t1
29:         t1 ← t2
30:         t2 ← t3
31:     end if
32:
33:     if t2.isConstatnt() then
34:         sub.add(t1 : t2)
35:         return sub
36:     end if
37:
38:     if t2.isFunction() then
39:         if t2.contains(t2) then
40:             retun error
41:         else
42:             sub.add(t1 : t2)
43:             return sub
44:         end if
45:     end if
46: end procedure
47:
48:
49: procedure UNIFICATION(L1, L2)
50:     sub ← new List()
51:     if L1.getPredicateSymbol()! = L2.getPredicateSymbol() then
52:         returnerror
53:     end if
54:
55:     t1Subterms = L1.getChilds()
56:     t2Subterms = L2.getChilds()
57:
58:     for i in t1Subterms.length do
59:         sub ← UnificationTerm(t1SubTerms[i], t2SubTerms[i], sub)
60:         if sub == error then
61:             return error
62:         end if
63:     end for
64:     return sub
65: end procedure
```

5.2.6.3 Konečnost algoritmu

- Protože literál je konečné délky, tak i jeho derivační strom je konečný.

- Algoritmus prochází derivační strom obou literálů jen jednou (prochází derivační strom do hloubky a každý uzel navštíví pouze jednou).
- V každém kroku porovná jestli se jedná o stejné termy a eventuálně vytvoří odpovídající substituci. Tento krok je v lineárním počtu operací a tudíž je konečný.

Algoritmus je konečný.

5.2.6.4 Časová složitost

Časová složitost je vztažená k délce delšího pozitivního literálu. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Algoritmus jednou projde derivační strom literálu do hloubky, při tom provede n kroků.
- Algoritmus v každém kroku porovná podstromy obou literálů. To je v lineárním čase ($\Theta(n)$).
- Ostatní operace jsou v konstantním čase ($\Theta(1)$)

Celkově tedy algoritmus provede n kroků a v každém kroku v nejhorším případě n operací.
Časová složitost je: $\Theta(n^2)$

5.2.7 Algoritmus vytvoření resolventy 2 klauzulí

5.2.7.1 Algoritmus

Algoritmus na vstupu předpokládá 2 klauzule.
Algoritmus vrátí buď resolventu klauzulí, nebo pokud ji nelze vytvořit tak *chyba*.

1. Algoritmus najde kandidáty na komplementární literály.
   To jsou literály, které mají stejný predikátový symbol a v jedné klauzuli je pozitivní a v druhé jako negativní.
2. Na každou dvojici kandidátů na komplementární literály zavolá unifikační algoritmus.
   (a) Pokud unifikační algoritmus nalezne unifikaci, tak je unifikace aplikovaná na obě klauzule.
   Pokud unifikace není nalezena, tak algoritmus pokračuje další dvojicí literálů.
   (b) Poté je z každé klauzule odebrán aktuální literál a obě klauzule jsou spojeny pomocí disjunkce.
   (c) Algoritmus vrátí nově vytvořenou klauzuli a končí.
3. Pokud nelze unifikovat žádnou dvojici kandidátů na komplementární literály, tak algoritmus vrátí chybu a skončí.

5.2.7.2 Pseudokód

```
1: procedure CreateResolvent(clause1, clause2)
2:     for [lit1, lit2] in getCandidateForComplementarLiteral(clause1, clause2) do
3:         unification ← Unification(lit1, lit2)
4:         if unification == error then
5:             continue
6:         end if
7:
8:         clause1.removeLiteral(lit1)
9:         clause2.removeLiteral(lit2)
10:        clause1.applyUnification(unification)
11:        clause2.applyUnification(unification)
12:
13:        return crateDisjunction(clause1, clause2)
14:    end for
15:
16:    return error
17: end procedure
```

5.2.7.3 Konečnost algoritmu

- Algoritmus vytvoří kombinaci každého literálu z jedné klauzule s každým literálem z druhé klauzule.
  Vzhledem k tomu že počet klauzulí je konečný tak i vytvoření všech kombinací je v konečném počtu kroků.
  V každém kroku algoritmus ověří jestli mají 2 literály stejný predikátový symbol a mají opačné znaménko. To je v konečném počtu operací.
- Algoritmus dále iteruje přes všechny vzniklé kombinace, kterých je konečný počet.
- Na každou kombinaci je zavolán algoritmus pro unifikaci literálů který je konečný.
- Aplikování nalezené unifikace a sloučení 2 klauzulí je v konečném počtu operací.
  Aplikování unifikace spočívá v průchodu celého derivačního stromu a nahrazení jednotlivých termů.

Algoritmus je konečný.

5.2.7.4 Časová složitost

Časová složitost je vztažená k délce delší klauzule. Nebo-li k počtu uzlů v jejím derivačním stromu.

- Algoritmus vytvoří kombinaci každého literálu z jedné klauzule s každým literálem z druhé klauzule. Vytvoření kombinací má časovou složitost $\Theta(n^2)$
- Algoritmus dále iteruje přes všechny vzniklé kombinace, kterých je n^2.

- Na každou kombinaci je zavolán algoritmus pro unifikaci literálů který běží v čase $\Theta(n^2)$.
- Aplikování nalezené unifikace a sloučení 2 klauzulí je v lineárním čase. ($\Theta(n)$)

Algoritmus provede n^2 kroků a v každém kroku provede $\Theta(n^2)$ operací. Celková složitost algoritmu je $\Theta(n^4)$.

5.2.7.5 Heuristika

Vzhledem k velké časové náročnosti algoritmu, je dobré se zamyslet nad heuristikou, která by v některých případech urychlila výpočet.

Vhodná heuristika by měla co nejlépe predikovat, které dva literály budou komplementární, eventuálně které 2 nebudou komplementární. Tím pádem by se minimalizoval počet generovaných a zkoumaných kandidátů na komplementární literály.

Dále lze dosáhnout zrychlení generovaní kombinací literálů dynamicky, aby algoritmus nemusel vytvářet všechny kombinace.

Jedno z možných řešení je z jedné klauzule si za-indexovat do slovníku literály kde klíč bude kombinace predikátového symbolu a příznaku jestli se jedná o kladnou nebo zápornou klauzuli. Poté může algoritmus procházet literály z druhé klauzule a pro každý literál si najít ve slovníku vhodné kandidáty, což v ideálním případě sníží složitost z $\Theta(n^4)$ na $\Theta(n^3)$ (Jedná se o případ, kdy se bude každý predikátový symbol vyskytovat v klauzuli v řádu jednotek. Jejich počet můžeme tím pádem považovat za konstantní - amortizovaná složitost vyhledání a zapsání do slovníku je v konstatním čase.)

5.2.8 Factoring algoritmus

Algoritmus, se pokusí zjednodušit klauzuli, a to pomocí sloučení literálů se stejným znaménkem které lze unifikovat.
Zároveň se algoritmus pokusí detekovat, jestli klauzule tautologie - obsahuje komplementární literály.
Algoritmus je obdobný jako pro vytváření resolvent.

5.2.8.1 Algoritmus

Vstupem algoritmu je klauzule a výstupem je upravená klauzule nebo *true*. *True* značí, že klauzule je tautologie.

1. Algoritmus najde kandidáty na redukovatelné literály.
   To jsou literály, které mají stejný predikátový symbol.
2. Na každou dvojici kandidátů na redukovatelné literály zavolá unifikační algoritmus.
   (a) Pokud unifikační algoritmus nalezne unifikaci, tak je unifikace aplikovaná na klauzuli.
   (b) Pokud mají oba literály stejné znaménko, je z každé klauzule odebrán nalezený a unifikovaný literál, algoritmus pokračuje procházením kombinací literálů.

(c) Pokud mají literály opačné znaménko, jedná se o tautologii a algoritmus vrací *true* a skončí.

3. Algoritmus vrátí redukovanou klauzuli.

5.2.8.2 Pseudokód

```
1: procedure FACTORING(clause)
2:     for [lit1, lit2] in getCandidateForUnification(clause) do
3:         unification ← Unification(lit1, lit2)
4:         if unification == error then
5:             continue
6:         end if
7:
8:         if (not lit1.isPositiveLiteral()) = lit2.isPositiveLiteral() then
9:             return true
10:        end if
11:        clause.removeLiteral(lit1)
12:        clause.applyUnification(unification)
13:    end for
14:
15:    return clause
16: end procedure
```

5.2.8.3 Konečnost algoritmu

- Algoritmus vytvoří kombinaci každého literálu s každým.
  Vzhledem k tomu že počet klauzulí je konečný tak i vytvoření všech kombinací je v konečném počtu kroků.
  V každém kroku algoritmus ověří jestli mají 2 literály stejný predikátový symbol. To je v konstantním počtu operací.
- Algoritmus dále iteruje přes všechny vzniklé kombinace, kterých je konečný počet.
- Na každou kombinaci je zavolán algoritmus pro unifikaci literálů který je konečný.
- Aplikování nalezené unifikace a odebrání klauzule je v konečném počtu operací.

Algoritmus je konečný.

5.2.8.4 Časová složitost

Časová složitost je vztažená k délce klauzule. Nebo-li k počtu uzlů v jejím derivačním stromu.

- Algoritmus vytvoří kombinaci každého literálu s každým.
- Nalezení jednotlivých literálů je v lineárním počtu operací ($\Theta(n)$)

- Nalezení kandidátů na redukovatelné literály je v kvadratickém počtu operací ($\Theta(n^2)$) Algoritmus porovná každý literál s každým.
- Algoritmus dále iteruje přes všechny vzniklé kombinace, kterých je n^2.
- Na každou kombinaci je zavolán algoritmus pro unifikaci literálů který běží v čase $\Theta(n^2)$.
- Aplikování nalezené unifikace a odebrání klauzule je v lineárním čase. ($\Theta(n)$)

Algoritmus provede n^2 kroků a v každém kroku provede $\Theta(n^2)$ operací. Celková složitost algoritmu je $\Theta(n^4)$.

5.2.8.5 Heuristika

Vzhledem k velké časové náročnosti algoritmu, je dobré se zamyslet nad heuristikou, která by v některých případech urychlila výpočet.

Vhodná heuristika by měla co nejlépe predikovat, které dva literály lze unifikovat, eventuálně které 2 nelze unifikovat. Tím pádem by se minimalizoval počet generovaných a zkoumaných kandidátů na redukovatelné literály.

Jedno z možných řešení je z klauzule si za-indexovat do slovníku literály, kde klíč bude predikátový symbol. Poté může algoritmus procházet pouze literály se stejným predikátovým symbolem. To v ideálním případě sníží složitost z $\Theta(n^4)$ na $\Theta(n^3)$ (Jedná se o případ, kdy se bude každý predikátový symbol vyskytovat v klauzuli v řádu jednotek. Jejich počet můžeme tím pádem považovat za konstantní.)

5.2.9 Rezoluční algoritmus

5.2.9.1 Algoritmus

Vstupem algoritmu je list klauzulí.
Výstupem algoritmu je množina resolvent pokud algoritmus skončí a nevygeneruje prázdnou resolventu (množina klauzulí je splnitelná). Pokud algoritmus vygeneruje prázdnou resolventu, tak vrátí false (množina klauzulí je nesplnitelná). Pokud algoritmus v daném časovém limitu nedoběhne vrátí chyba (algoritmu se nepodařilo určit jestli je formule splnitelná).

1. Algoritmus náhodně vybere 2 klauzule, které mezi sebou ještě nezkoušel kombinovat a zkusí vytvořit resolventu.
2. Pokud se podaří vytvořit resolventu

5.2.9.2 Pseudokód

```
1: procedure RESOLUTION(listClauses)
2:     while hasNextCombination(listClauses) do
3:         cl1, cl2 ← nextCombination(listClauses)
4:         resolvent ← CreateResolvent(cl1, cl2)
```

```
5:        if isEmpty(resolvent) then
6:            return false
7:        end if
8:
9:        if isError(resolvent) then
10:           continue
11:       end if
12:
13:       resolvent ← factoring(resolvent)
14:       if resolvent == true then
15:           continue
16:       end if
17:   end while
18:
19:   return true
20: end procedure
```

5.2.9.3 Konečnost algoritmu

Algoritmus není konečný.

Jako příklad kdy algoritmus nikdy neskončí uvedu výpočet rezoluce nad množinou dvou klauzulí.

$\{P(x) \vee P(f(x)); \neg P(f(a))\}$

kde:
P je predikátový symbol
f je funkční symbol
x je proměnná
a je konstanta

C_1: $P(x) \vee P(f(x))$
C_2: $\neg P(f(a))$

$R_1[C_1, C_2]$: $P(f(f(a)))$
$R_2[R_1, C_1]$: $P(f(f(f(a))))$
$R_3[R_2, C_1]$: $P(f(f(f(f(a)))))$
....
$R_n[R_{n-1}, C_1]$: $P(f(f(f(....f(a)....))))$

Algoritmus nejdříve vytvoří resolventu z dvou vstupních klauzulí.
Poté vždy vytvoří resolventu z předchozí resolventy a první klauzule. Při unifikování se donekonečna rozrůstá zanoření funkce f.

Některé z heuristik se snaží tomuto problému zabránit, ale úplně mu nelze zabránit. Více v kapitole o heuristikách 5.2.9.5

5.2.9.4 Časová složitost

Vzhledem k tomu, že algoritmus není konečný, nelze určit časovou složitost.

5.2.9.5 Heuristika

Existuje mnoho různých heuristik v jakém pořadí a jaké klauzule vybrat. Já zde uvedu dvě základní a to procházení do hloubky a do šířky. Dále zde rozeberu facotring jako jednu z optimalizačních metod pro resoluci.

Procházení do hloubky Na vstupu je seřazený seznam klauzulí. Algoritmu bera jednotlivé klauzule v daném pořadí a vytváří z nich resolventy. Resolventu vytváří vždy z poslední klauzule a z jedné, ze vstupních klauzulí dokud nedostane kontradikci nebo dokud lze vytvářet další resolventy.

Tento postu často vede k Rychlému nalezení kontradikce, má však nevýhodu, že se může lehce zacyklit (viz příklad u popisu konečnosti).

Procházení do šířky Algoritmus vytvoří všechny možné resolventy ze vstupních klauzulí. Poté přidá přidá ke vstupním klauzulím nové resolventy a zase z tohoto seznamu vytvoří všechny možné resolventy. a tento postup opakuje dokud nedostane kontradikci nebo dokud lze vytvářet další resolventy.

Tento postup je sice často pomalejší, na druhou stranu má výhodu, že více předchází cyklení (viz příklad u popisu konečnosti).

Factoring Factoring není heuristika, která by pomáhala zvolit vhodné klauzule k resoluci, ale metoda pro zmenšení velikosti klauzulí a tím i zmenšit celkový počet potřebných resolvent k nalezení kontradikce.

Facotoring může v některých případech pomoci předejít cyklení (viz příklad u popisu konečnosti).

5.2.10 Algoritmus pro ověření Subsumpce

Pro ověření subsumpce využiji vztah pro převedení sémantického důsledku na nesplnitelnost formule 3.5.3 Základní vztahy - 6. Po aplikování tohoto vztahu stačí použít algoritmus pro splnitelnost formule (Rezoluční algoritmus 5.2.9)

5.2.11 Least general generalization algoritmus

Popis tohoto algoritmu vychází z *Unification and anti-unification, Erik Jcobsen*[8] a *Symbolic Machine Learning, Filip Železný, Jiří kléma*[6]
Algoritmus pro nalezení least general generalization dvou literálů. Algoritmus hledá pro každý literál takovou substituci, která by jej zobecňovala a zároveň po jejich aplikování se budou literály rovnat.

5.2.11.1 Algoritmus

Vstupem jsou dva literály se stejným predikátovým symbolem a výstupem generalizace těchto 2 literálu.

1. Algoritmus prochází postupně strom obou literálu do hloubky.
2. Pokud se podstromy těchto dvou literálu nerovnají tak:
   (a) Algoritmus zkontroluje jestli jsou oba termy funkce stejné funkce.
   Pokud jsou algoritmus zavolá sám sebe na všechny dvojice parametrů a vrátí novou zobecněnou funkci.
   (b) Algoritmus zkontroluje jestli existuje pro aktuální termy substituce.
   Tím je myšleno, že existuje proměnná v, kterou lze substituovat na oba termy.
   Pokud Existuje tak je aplikovaná a algoritmus vrátí výsledek substituce.
   (c) Pokud se aktuální podstromy nerovnají je vytvořena nová proměnná v a pro každý term t substituce $\{v : t\}$.
   algoritmus vrátí novou proměnnou v
3. Pokud se podstromy těchto dvou termů rovnají, algoritmus vrátí aktuální term.

5.2.11.2 Pseudokód

```
1: procedure AntiUnification(term1, term2, sub1, sub2)
2:     if lit1 == lit2 then
3:         return lit1
4:     end if
5:
6:     if term1.isFunction() and term2.isFunction() and term1 samefunctionas term2 then
7:         args ← []
8:         for i in term.args.length do
9:             res ← AntiUnification(term1.args[i], term2.args[i], sub1, sub2)
10:            args.append(res)
11:        end for
12:        retun term.newFunction(args)
13:    end if
14:
15:    for key in sub1.keys do
16:        if sub1[key] == term1 and sub2[key] == term2 then
17:            return key
18:        end if
19:    end for
20:    var = getNewVariable()
21:    sub1[var] ← term1
22:    sub2[var] ← term2
23:    return var
24: end procedure
```

5.2.11.3 Konečnost algoritmu

- Algoritmus projde jednou strom literálů a v každém kroku porovná rovnost podstromů
- Porovnání literálů je v konečném počtu operací
- Pokud se nerovnají, algoritmus zkusí aplikovat substituci a eventuálně vytvoří novou substituci.
  - Kontrola jestli existuje substituce je v konečném počtu operací. Jedná se o iterování přes všechny substituce.
  - Vytvoření substituce je v konečném počtu operací. Jedná se o přidání položky do slovníku.
  - Aplikování substituce je v konstantním počtu operací. Jedná se o nahrazení jednoho uzlu jiným.

Algoritmus je konečný.

5.2.11.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Algoritmus jednou projde derivační strom formule do hloubky, při tom provede n kroků.
- Algoritmu v každém kroku porovná rovnost podstromů - Porovnání podstromů je v lineárním čase.
- Pokud se nerovnají, algoritmus zkusí aplikovat substituci a eventuálně vytvoří novou substituci.
  - Kontrola, jestli existuje substituce je v lineárním počtu operací. Jedná se o vyhledání ve slovníku.
  - Vytvoření substituce je v lineárním počtu operací. Jedná se o přidání položky do slovníku.
  - Aplikování substituce je v konstantním počtu operací. Jedná se o nahrazení jednoho uzlu jiným.

Časová složitost je $\Theta(n^2)$

5.2.12 Paramodulation

Popis tohoto algoritmu je vychází z *AUTOMATED REASONING*[1], *Pramodulation*[2] a *Symbolic Logic and Mechanical Theorem Proving*[3]

Paramodulation je metoda pro aplikování přepisovacího pravidla pro rovnost(viz. kapitola 4.5). Její využití je hlavně v rezoluční metodě, pro logiku prvního řádu s rovností.

Vstupem algoritmu jsou dvě klauzule a výstupem je nová klauzule, nebo chyba pokud ji pomocí paramodulace nelze vytvořit.

5.2.13 Algortimus

1. Algoritmus projde strom obou klauzulí a zkontroluje jestli jsou ve správném tvaru.
2. Algoritmus zavolá unifikační algoritmus pro vytvoření MGU.
3. Algoritmus aplikuje MGU a poté aplikuje nahrazovací pravidlo

5.2.13.1 Pseudo kód

```
1: procedure PARAMODULATION(clause1, clause2)
2:     if clause2.containsEquality() then
3:         cl ← clause1
4:         clause1 ← clause2
5:         clause2 ← cl
6:     end if
7:
8:     if ¬clause1.containsEquality() then
9:         return error
10:    end if
11:
12:
13:    equality ← clause1.getEquality()
14:    unification = mgu(clause2, negation.getLeftChild()
15:    if ¬unification then
16:        return chyba
17:    end if
18:    clause1.remove(equality)
19:    clause1.substitute(unification)
20:    clause2.substitute(unification)
21:    equality.substitute(unification)
22:    clause2.substitute(equality.getLeftChild() : equality.getRightChild())
23:
24:    return merge(clause1, clause2)
25: end procedure
```

5.2.14 Zploštění klauzulí

Popis tohoto algoritmu je vychází z *New Techniques that Improve MACE-style Finite Model Finding, K. Claessen, N. Sörensson*[9] a *Hledání modelů v predikátové logice 1. řádu, Jiří Vyskočil*[14]
Vstupem algoritmu je klauzule a výstupem je plochá klauzule.

5.2.14.1 Algoritmus

1. Algoritmus prochází derivační strom formule do hloubky.

2. Pokud algoritmu narazí na nemělký literál tak můžou nastat tyto dva případy:

   (a) V klauzuli φ vyskytuje alespoň jeden podterm t, který není proměnná.
   Algoritmus podterm t v klauzuli φ nahradí novou proměnnou x a přidá do klauzule $x = t$
   Klauzule bude mít tedy tvar $\varphi' \vee x = t$

   (b) V klauzuli φ vyskytuje literál tvaru $x\neg = y$ (tedy předpokládejme, že φ je tvaru $\varphi' \vee x\neg = y$)
   V klauzuli φ' nahradíme všechny výskyty y za x a odebereme z klauzule φ literál $x\neg = y$

3. Algoritmus po projití celého stromu vrátí výslednou klauzuli.

5.2.14.2 Pseudo kód

```
1: procedure FlattenClause(clause)
2:     listOfLits = toList(clause, OR)
3:     for lit in listOfLits do
4:         if lit is equality or lit is inequality then
5:             if lit is inequality and lit.leftChild.isVar() and lit.rightChild.isVar() then
6:                 clause.remove(lit)
7:                 clause.substitute(lit.rightChild, lit.leftChild)
8:             end if
9:
10:            if lit is equality and ¬lit.leftChild.isVar() and ¬lit.rightChild.isVar() then
11:                var = getNewVariable()
12:                clause.addOR(EQUALITY(lit.rightChild, var))
13:                lit.rightChild = var
14:            end if
15:        else
16:            functions ← getFunctions(lit)
17:            for f in functions do
18:                var = getNewVariable()
19:                newLit ← FlattenClause(EQUALITY(f, var))
20:                clause.substitute(f, var)
21:                clause.addOR(INEQUALITY(f, var))
22:            end for
23:        end if
24:    end for
25: end procedure
```

5.2.14.3 Konečnost algoritmu

- Algoritmus projde jednou strom klauzule a v každém kroku zkontroluje jestli nenarazil na nemělký literál
- Kontrola mělkosti literálu je v konečném počtu operací

- Pokud narazí na nemělký literál tak použije jedno z nahrazovacích pravidel.
  Aplikace nahrazovacích pravidel je v konstantním počtu operací.
- Aplikace substituce je 1 projití stromu formule a nahrazení odpovídajícího výrazu.
  Nahrazení odpovídajícího výrazu je v konstantním počtu operací.

Algoritmus je konečný.

5.2.14.4 Časová složitost

Časová složitost je vztažená k délce formule. Nebo-li k počtu uzlů v jeho derivačním stromu.

- Algoritmus jednou projde derivační strom formule do hloubky, při tom provede n kroků.
- Algoritmu v každém kroku zkontroluje jestli nenarazil na nemělký literál.
  Kontrola mělkosti literálu je konečném počtu operací
- Pokud narazí na nemělký literál tak použije jedno z nahrazovacích pravidel.
  Aplikace nahrazovacích pravidel je v konstantním počtu operací.
- Aplikace substituce je 1 projití stromu formule a nahrazení odpovídajícího výrazu (n operací).
  Nahrazení odpovídajícího výrazu je v konstantním počtu operací.

Časová složitost je $\Theta(n^2)$

Kapitola 6

Návrh

6.1 Formát zápisu formule

Při návrhu zápisu formule jsem se snažil o:

- Co největší čitelnost zápisu formule.
- Podobnost matematickému zápisu formule.
- Konzistentnost se zápisem ostatních matematických výrazů.

Proměnné pro zápis proměnných a funkcí využiji stávající funkcionalitu.
Lze je tedy zapsat:

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> f = Function('f')
>>> f(x,x)
```

Návrh zápisu konstant vychází ze zápisu proměnných.
Například:

```
>>> from sympy import FolConstat
>>> ann = FolConstat('ann')
```

Návrh zápisu predikátu vychází ze zápisu nedefinovaných matematických funkcí. Oproti nedefinovaným matematickým funkcím.
Například:

```
>>> from sympy import FolPredicate
>>> P = FolPredicate('P')
>>> formula = P(f(ann), x)
```

U zápisu logických operandů vycházím ze zápisu formulí výrokové logiky, kterou knihovna podporuje.

- Negace ($\neg$) se zapisuje jako $\sim$
- Konjunkce ($\wedge$) se zapisuje jako &
- Disjunkce ($\vee$) se zapisuje jako |
- Implikace ($\Rightarrow$) se zapisuje jako >>
- Implikace ($\Leftarrow$) se zapisuje jako <<
- Ekvivalence ($\Leftrightarrow$) nelze přímo zapsat, ale musí se vytvořit ručně: $FolEquivalency(leftFormula, rightFormu$
- Rovnost ($=$) nelze přímo zapsat, ale musí se vytvořit ručně: $FolEquality(leftFormula, rightFormula)$

Například:

```
>>> Q(x) | Z(y) & P(f(x), z)
```

Náročnější byl návrh zápisu obecných a existenčních kvantifikátorů. V matematickém zápisu se zapisují $\exists x \forall y P(f(y), x))$. ale v Pythonu nelze zapsat něco na způsob

```
>>> Exist(x) ForAll(y) P(f(x), y)
```

Protože mezi jednotlivými objekty/proměnnými musí být nějaký operátor, který je spojí dohromady. Proto jsem zkoušel hledat vhodný operátor, kterým by šlo připojit k kvantifikátoru formuli. Napadla mě tečka (například v PHP se využívá ke spojení dvou řetězců), ale tu v Pythonu lze použít pouze k přístupu k metodám nebo vlastnostem objektu. Proto jsem uvažoval o operátoru $+$, ten by sice syntakticky šel použít, ale zápis

```
>>> Exist(x) + ForAll(y) + P(f(x), y)
```

mi nepřijde hezký. Po projití všech možných operátorů jsem zjistil, že žádný se pro tento zápis nehodí. Ale napadla mě možnost, že konstruktor objektu kvantifikátoru by přijímal 2 parametry. První parametr je proměnná, ke které se kvantifikátor váže. Druhý parametr je formule, na kterou je kvantifikátor aplikován.

```
>>> Exist(x, ForAll(y, P(f(x), y)))
```

Tentno zápis mi pořád nepřišel ideální, obzvláště v případě, že bych chtěl zapsat nějakou větší formuli.

```
>>> Exist(x, Q(x) | Exist(y, Z(y) & ForAll(z, P(f(x), z))))
```

Nakonec mě napadlo, že v Pythonu lze vytvořit objekt, který je callable, a lze ho volat stejně jako funkci.

```
>>> Exist(x) (ForAll(y) (P(f(x), y)))
```

Tento zápis mi v rámci možností co poskytuje Python přijde nejlepší, nejpřehlednější a nejvíce podobný matematickému zápisu. Zároveň je konzistentní se zápisem matematických výrazů ve zbytku knihovny. Příklad složitější formule
Matematický zápis (pokud závorkujeme formuli aby bylo jasné co se k čemu váže)
$\exists x(Q(x) \vee \exists y(Z(y) \wedge \forall z(P(f(x), z))))$
Mnou navrhovaný zápis v Pythonu

```
>>> Exist(x) (Q(x) | Exist(y) (Z(y) & ForAll(z) (P(f(x), z))))
```

6.2 Začlenění do modulů SymPy

Knihovna Sympy obsahuje modul logic, který umožňuje pracovat s výrokovou logikou. Proto jsem uvažoval jestli nerozšířit tento modul i o logiku prvního řádu, protože na první pohled mi přišlo. že logika prvního řádu rozšiřuje výrokovou logiku.

Při detailnějším studování modulu logic jsem zjistil, že rozšíření tohoto modulu o logiku prvního řádu nedává smysl. Jednalo by se o samostatnou funkcionalitu nebo bych funkcionalitu složitě a nehezky rouboval na funkcionalitu výrokové logiky.

Z tohoto důvodu jsem se rozhodl pro nový modul, který jsem nazval first_order_logic.

Další z kroků může být, přepsat existující modul logic, tak aby využíval můj nový modul. Tato varianta mi nepřijde jako vhodná, protože by se pravděpodobně narazilo na stejné problémy jako při využití modulu logic jako základ pro logiku prvního řádu. Tato varianta by ale do budoucna potřebovala důsledně prozkoumat a i zvážit jestli výhody této změny převýší nevýhody.

6.3 Datový model

Základní datové objekty jsou:

- Proměnné, které jsou reprezentovány třídou *Symbol*
- Konstanty, které jsou reprezentovány třídou *FOLConstant*
- Funkce, které jsou reprezentovány třídou *Function*
- Predikáty, které jsou reprezentovány třídou *Predicate*
- Logické operátory:

  - Negace, která je reprezentovaná třídou *FOLNot*
  - Konjunkce, která je reprezentovaná třídou *FOLAnd*
  - Disjunkce, která je reprezentovaná třídou *FOLOr*
  - Implikace, která je reprezentovaná třídou *FOLImplies*
  - Ekvivalence, která je reprezentovaná třídou *FOLEquivalent*
  - Rovnost, která je reprezentovaná třídou *FOLEquality*
    Rovnost najezdil od ostatních logických operátorů není reprezentovaná jako logický operátor, ale jako speciální typ predikátu. Toto řešení jsem zvolil z důvodu, že práce s rovností je výrazně bližší práci s predikáty než s logickými operátory.

- Kvantifikátory:
  - Obecný kvantifikátor, který je reprezentovaná třídou *Forall*
  - Existenční kvantifikátor, který je reprezentovaná *Exists*

Logické operátory, kvantifikátory a predikáty mají přetížené operátory pro metody pro logické operace ($\sim$, |, &, >>)

6.4 Struktura tříd

6.4.1 Konstanty

Třída Třída *FOLConstatnt* rozšiřuje třídu *Symbol* a jediná její funkce je přidání příznaku, že se jedná o konstantu a ne o proměnnou.

6.4.1.1 Predikáty operátory

Predikáty jsou potomkem třídy *Function*, využívají její způsob dynamického vytváření nových tříd.

Zápis:

```
>>> P = FOLPredicate('P')
```

totiž vytvoří novou třídu P, která je potomkem třídy *FOLPredicate* a až volání

```
>>> P(x,y)
```

vytvoří instanci třídy.

6.4.1.2 Kvantifikátory

Obdobně jako predikáty, jsou kvantifikátory potomkem třídy *Function*, využívají její způsob dynamického vytváření nových tříd.

6.4.2 Logické operátory

Logické operátory jsou také potomkem třídy *Function*, narozdíl od predikátů a kvantifikátorů nevyužívají dynamické vytváření nových tříd. Ale vytváří se rovnou jejich instance.

6.4.3 Algoritmy

Jednotlivé algoritmy jsou samostatné funkce, které jsou ve složce *algorithm* v modulu $first_order_logic$ a jsou rozdělené do jednotlivých souborů.

Kapitola 7

Implementace

Většina implementace je popsaná v předchozích kapitolách, proto se zde zmíním pouze v kterých kapitolách je lze najít a popíšu proč a jaké heuristiky jsem u algoritmů implementoval.

7.1 Začlenění do knihovny SymPy

Začlenění do knihovny je popsané v kapitole Návrh 6, jedná se o kapitoly "Začlenění do modulů SymPy"6.2 a "Struktura tříd"6.4.

7.2 Datový model

Datový model je popsaný v kapitole Návrh 6, jedná se hlavně o kapitolu "Datový model"6.3 a částečně v kapitole "Struktura tříd"6.4.

7.3 Implementace algoritmů

Algoritmy jsou popsaný slovně v kapitole "Analýza - Algoritmy"5.2. Každý algoritmu obsahuje slovní popis a pseudokód.

Jejich struktura je popsaná v kapitole "Návrh - Struktura tříd"6.4

7.4 Implementované heuristiky

Jediná optimalizační metoda která byla implementovaná, je factoring u vytváření resolvent. Popis této metody je uveden v Analýze.

Kapitola 8

Testování

8.1 Automatické testování

Pro testování jsem využil testovací framework, který je součástí Sympy.
Testy jsou v souboru $first_order_logic/testis/test_folalg.pyajednotlivtestyjsourozdlenpodlefunkcionality$:

Přetěžování operátorů

Převod formule do negativní normální formy

Převod formule do prenexní normální formy

Převod formule do skolemovi normální formy

Převod formule na množinu klauzulí

Unifikace dvou termů

Vytváření resolventy z dvou klauzulí

Factoring

Rezoluční metoda

Rezoluční metoda s rovností

Zploštění klauzulí

Least general generalization

Při vytváření testovacích dat jsem vycházel z příkladů použitých v článcích/knihách z kterých jsem i čerpal popis algoritmů. Zároveň jsem se snažil pokrýt co největší množství různých vstupních dat.

Testy jsou převážně koncipovány tak, že tetují nejdříve základní a data a poté nějaké pokročilejší.

Netestoval jsem nevalidní data, protože současná implementace nekontroluje vstupní data a proto nemá smysl testovat nekorektní vstupní data.

Všechny napsané testy úspěšně procházejí.

Kapitola 9

Závěr

9.1 Splnění cílů

Algoritmy všechny popsané v zadání, jsem analyzoval a implementoval, tak aby mohli být zařazeny do knihovny SymPy jako nový modul. Co jsem bohužel nestihl, je vytvoření dokumentace pro tento nový modul. Dále by bylo vhodné implementovat, heuristik pro tvorbu resolvent.

9.2 Možné rozšíření

Jedno z hlavních rozšíření je implementovat heuristiky pro tvorbu resolvent.

Dále by bylo vhodné analyzovat možnost vytvořit abstraktní modul pro logiku, z kterého by vycházela, výroková logika, logika prvního řádu a do budoucna třeba logika vyššího řádu. Tato změna, by ale znamenala citelný zásah do struktury celé knihovny a proto tomu musí předcházet důkladná analýza a diskuze s ostatními přispěvateli do této knihovny.

V neposlední řadě by bylo vhodné rozšířit počet implementovaných algoritmů o další, například *Převod problému hledání modelu na SAT*.

Literatura

[1] *Automated Reasoning* [online]. [cit. 1. 5. 2019]. Dostupné z: <https://www.doc.ic.ac.uk/~kb/MACTHINGS/SLIDES/2013Notes/12LEQR4up13.pdf>.

[2] *Paramodulation* [online]. [cit. 1. 5. 2019]. Dostupné z: <http://profs.sci.univr.it/~farinelli/courses/ar/slides/paramodulation.pdf>.

[3] CHIN-LIANG CHANG, R. C.-T. L. *Symbolic Logic and Mechanical Theorem Proving.* 111 Fifth Avenue, New York, New York 10003 : Academic press, INC.

[4] DEMLOVá, M. *Matematical logic.* Zikova 4, 166 36 Praha 6 : Vydavatelství ČVUT, 1st edition, 1999.

[5] PYTHON DOKUMENTACE, P. [online]. [cit. 1. 5. 2019]. Dostupné z: <https://docs.python.org/3/library/doctest.html>.

[6] FILIP ŽELEZNý, J. k. *Symbolic Machine Learning* [online]. [cit. 1. 5. 2019]. Dostupné z: <https://cw.fel.cvut.cz/old/_media/courses/b4m36smu/smu-textbook.pdf>.

[7] HUDAK, P. Domain-specific languages. In *Handbook of programming languages, vol. III: little languages and tools, chapter 3*, s. 39–60, 1998.

[8] JACOBSEN, E. *Unification and anti-unification* [online]. [cit. 1. 5. 2019]. Dostupné z: <http://erikjacobsen.com/pdf/unification.pdf>.

[9] K. CLAESSEN, N. S. New Techniques that Improve MACE-style Finite Model Finding. Workshop: Model Computation - Principles, Algorithms, Applications, 2003.

[10] M. DEMLOVá, B. P. *Matematická logika.* Zikova 4, 166 36 Praha 6 : Vydavatelství ČVUT, 1st edition, 1997.

[11] MEURER, A. et al. *SymPy: symbolic computing in Python* [online]. 2017. [cit. 1.5.2019]. Dostupné z: <https://peerj.com/articles/cs-103/>.

[12] Přispěvatelé SymEngine. *SymEngine* [online]. [cit. 1. 5. 2019]. Dostupné z: <https://github.com/symengine/symengine>.

[13] Přispěvatelé SymPy. *Projects SymPy* [online]. [cit. 1. 5. 2019]. Dostupné z: <https://www.sympy.org/en/index.html>.

[14] VYSKOčIL, J. *Hledání modelů v predikátové logice 1. řádu* [online]. [cit. 1. 5. 2019]. Dostupné z: <https://cw.fel.cvut.cz/old/_media/courses/a4m33au/13-model_finding_methods.pdf>.

Příloha A

Obsah přiloženého CD

```
   .
FIlipSamanek.pdf
sympy
   sympy
       abc.py
       algebras
       assumptions
       benchmarks
       calculus
       categories
       codegen
       combinatorics
       concrete
       conftest.py
       core
       crypto
       deprecated
       diffgeom
       discrete
       external
       first_order_logic
          algorithm
             flattening.py
             heuristic
                resolution
                    basic.py
                    __init__.py
                    __pycache__
                        basic.cpython-36.pyc
                        __init__.cpython-36.pyc
             __init__.py
             least_general_generalization.py
```

```
            normal_form.py
            __pycache__
               flattening.cpython-36.pyc
               __init__.cpython-36.pyc
               least_general_generalization.cpython-36.pyc
               normal_form.cpython-36.pyc
               resolution.cpython-36.pyc
            resolution.py
         folalg.py
         __init__.py
         tests
             __init__.py
             run_tests.py
             test_folalg.py
      functions
      galgebra.py
      geometry
      holonomic
      __init__.py
      integrals
      interactive
      liealgebras
      logic
      matrices
      multipledispatch
      ntheory
      parsing
      physics
      plotting
      polys
      printing
      __pycache__
      release.py
      sandbox
      series
      sets
      simplify
      solvers
      stats
      strategies
      tensor
      this.py
      unify
      utilities
      vector
tex
```

```
    chapter1-Introduction.tex
    chapter2-SympyLibrary.tex
    chapter3-FirstOrderLogic.tex
    chapter4-FirstOrderLogicWithEquality.tex
    chapter5-Analysis.tex
    chapter6-Design.tex
    chapter7-Implementation.tex
    chapter8-Testing.tex
    chapter9-Conclusion.tex
    csplainnat.bst
    figures
       derivationTree.png
       LogoCVUT.eps
       LogoCVUT.pdf
       seznamcd.eps
       seznamcd.pdf
    FilipSamanek.tex
    hyphen.tex
    k336_thesis_macros.sty
    obrazky.tex
    reference.bib
    zav_prace.pdf
 tmp

52 directories, 48 files
```